ıTION PAGE

Form Approved
OMB No. 0704-0188

1

verage 1 hour per response, including the time for reviewing instructions, searching existing data sources, the collection of information. Send comments regarding this burden estimate or any other aspect of this to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
30 Dec 91

3. REPORT TYPE AND DATES COVERED
Research - FY90-92

4. TITLE AND SUBTITLE
DYNAMIC TERRAIN

5. FUNDING NUMBERS
$250K

6. AUTHOR(S)
Dr. M. Moshell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Simulation & Training (IST)
University of Central Florida (UCF)
12424 Research Parkway
Orlando, FL 32826

8. PERFORMING ORGANIZATION REPORT NUMBER
IST
TR-92-11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
PM TRADE (now STRICOM)
12350 Research Parkway
Orlando, FL 32826-3276

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
Contr. N61339-90-C-0041

11. SUPPLEMENTARY NOTES
This report is 1/2 of task under Ref. Contract, Head Mounted Displays (HMD) makes up the other half of the contract.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unlimited Distribution

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research deals with battlefield terrain modified after the program has begun. It demo's, for example, a tractor plowing ground, changing the terrain in real time. Also, a fluid flow study demo'd a tractor breaching a dammed fluid pond and shows the flow to the lowest elevation.

14. SUBJECT TERMS
berms, craters, ditches, flowing water, vehicle track marks, complex motion of rigid objects

15. NUMBER OF PAGES
1

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UL

INSTITUTE FOR SIMULATION AND TRAINING

Dynamic Terrain

Final Report

The body of this report is VSL Document VSLM91.40. This document is the Final Report for Contract N61339-90-C-0041 Task 1; Dynamic Terrain, sponsored by the Army Project Manager for Training Devices (PM TRADE).
All opinions herein expressed are solely those of the authors.

Contract N61339-90-C0041
PM TRADE

December 30, 1991

Visual Systems Laboratory
IST-TR-92-11

Prepared by

J. Michael Moshell, Project Manager

Xin Li, Curtis Lisle, Mikel Petty, Julie Carrington, Jennifer Burg,
Charles Hughes, Charles Campbell, Larry Gibbs, Kien Hua,
Rao Vempaty, Jinxiong Chen, Brian Blau

Reviewed by

Brian Goldiez

Institute for Simulation and Training • 12424 Research Parkway, Suite 300 • Orlando, Florida 32826

University of Central Florida • Division of Sponsored Research

Dynamic Terrain
CONTENTS

APPENDICES

Dynamic Terrain
Final Report

Abstract

This Report is a summary of work in 1990 and 1991 by the staff of the IST Visual Systems Laboratory on the subject of the realtime visualization of dynamic terrain. Most existing visual training simulators have a static terrain database. Real warfare involves extensive modification to the terrain, as vehicles move about, leave tread marks and destroy ground cover. Defensive emplacements and anti-tank obstacles are constructed; drainage ditches are prepared. Water crossings require the reduction of streambanks.

This Project is based upon the hypothesis that it is possible to construct networked realtime simulators which incorporate useful simulations of these phenomena, without markedly increasing the cost of the associated computing equipment.

The work of this Project has consisted of examining the inherent computational requirements of various phases of this simulation work, looking for opportunities to simplify models without sacrificing plausible behavior, and examining the implications of the new computational tasks on the underlying architecture of the networked simulators.

1. Overview

The Dynamic Terrain Project grew from a previous IST study, funded in 1989 by PM-TRADE, to investigate the feasibility of increasing the realism of graphical simulations. That study (Moshell 89) explored basic issues in the underlying technology and served to chart the course of the following research. In 1990, a specific group of features were selected for study. These included

- earthworks (berms, ditches, craters)
- complex motions of rigid objects (trees, logs, rocks)
- vehicle track marks
- flowing water

These features were used as test cases to drive the investigation of underlying graphical technology in four areas:

- visualization (generation of realistic polygon-based images)
- dynamics (making soil and water behave realistically)
- databases (representing a changing world in networked simulators)
- networking (communicating about changes to the database)

This report is organized in four major sections to cover the four technology areas listed above. We have tried to keep this report relatively brief, and to provide extensive backup documentation in the form of appendices. Most of these consist of internal reports that were written in the course of the research, and published papers.

2. Visualization

2.1 Goals.

There are a number of issues involved in visualizing terrain while it is undergoing changes. Three salient issues are

- terrain profile generation during excavation
- polygon budget management
- animation *versus* simulation

The following sections address each of these issues and describe the associated research.

2.2. Terrain Profile Generation

The following four software modules are all described in more detail in Appendix C.

The Terrain Editor. The first need was for a convenient source of realistic artificial terrain, so that subsequent experiments would not take place on a simple flat grid. An interactive editor was built, which used the Silicon Graphics Iris workstation and mouse. A 50 x 50 meter terrain patch was rendered as a wireframe, and a red moving cursor indicates the mouse's position. The left mouse button raised the terrain, the right lowered it.

Adjacent posts are raised by a spline extrapolation technique so as to maintain continuity.

Car Animation (Surface Travel). As an intermediate step in building the following (bulldozer) simulation, a simulated automobile was built to drive over the terrain. Its motion is controlled by the mouse. This software extracts the appropriate vehicle orientation and position from adjacent elevation posts.

The Uniform-Mesh Bulldozer. The bulldozer is a combination of techniques from the terrain editor and the moving car. The dozer blade serves as a local forcing function to set the heights of the terrain. Excess terrain volume which is "scraped off" by the moving blade is added to the moving berm in front of the blade, which is then smoothed by the bidirectional spline algorithm originally developed for the terrain editor.

The dozer simulation is purely kinematic. That is, no forces are computed. The dozer's treads do not slip, and the soil does not slump when the dozer leaves. Nevertheless the demonstration is surprisingly realistic.

The SIMNET-Terrain Bulldozer. The first dozer used only a uniform rectangular terrain mesh. The SIMNET system uses a technique called microterrain, whereby a large polygon is dissected into small triangular patches when detailed relief is needed. In this demo, the initial terrain is loaded from actual SIMNET terrain data. As the dozer moves into a terrain quadrangle, the quad is dissected into small triangles which are immediately edited by the moving blade.

2.3. Visualization Studies

An analytical study was performed to determine how many polygons would be needed for the rendering of typical dynamic terrain scenes. Using the MultiGen CAD tool, typical scenarios were constructed and examined. The broad conclusion of the study was that dynamic terrain at a level of fidelity appropriate for the CCTT (follow-on to SIMNET) system would require approximately twice the polygon capacity of the basic CCTT terrain requirement.

The study, with photographs, is provided in two papers which are attached as Appendix D.

2.4 The ANIM Animation System

To support the convenient visualization of moving objects, the ANIM software system was developed using the Silicon Graphics Iris. ANIM can accept an ASCII file containing a script which describes the motions of any number of polyhedral bodies, and produce a smooth animation. ANIM can also accept certain physical descriptions of objects and cause them to move according to the resulting constraints and forces.

This software has proven particularly useful in studies of physical simulation (described in the next section) and also in a collateral PM-TRADE sponsored project concerning visual databases. ANIM is described by the document in Appendix E.

3. Dynamics

3.1 Goals. A major component of the Dynamic Terrain project concerned the acquisition of state-of-the-art knowledge in realtime physical modeling, and its extension for specific problems. Within the last four years, a substantial body of literature has developed on this subject. The goals for the physical modeling component of this project were

- to explore the complexities of interactive physical modeling;
- to write software which implements some of the most promising techniques, so they could be evaluated for use in realtime simulation;
- to develop new techniques and advance the state of the art when deficiencies are discovered.

3.2 Constraints. The intellectual kernel of most physical modeling work is based on the concept of *constraints*. A constraint is a relationship between two variables which must remain true. For instance, a tank is constrained to remain on the ground by gravity. This fact could be represented by a constraint of the form

z(tank) = z(terrain(x(tank), y(tank)))

which says that the tank's z coordinate (height) equals the z coordinate of the terrain at the (x,y) location of the tank. This is an example of a *one-way constraint*, because it is not logical to require that the terrain's height change to match that of the tank. If some large force lifted the tank, the terrain would not necessarily follow.

A two-way constraint would be exemplified by the relationship between a tank and a tank recovery vehicle. Each exerts forces on the other, and the proper depiction of motion would require that the equations of motion be solved so that the acceleration of each object is determined by the sum of forces on it.

A constraint resolution system is a software system which solves such problems. It differs from a simple numerical integration of the differential equations (DEs) in that the problem may not actually be expressed in DEs; the constraint system may have to logically construct the DEs, depending on the relationships between objects.

As part of the DT project in 1990, a comprehensive survey of constraint mechanisms was prepared. We also reviewed the significant papers in the literature on physical modeling for graphics. Summary papers from these reviews are provided as Appendix F.

3.3. The PM Physical Modeling System. In conjunction with ANIM, a practical constraint system was implemented which uses the concept of restoring forces to model joints. For instance, a chain of links is modeled as a set of rigid bodies coupled by springs. PM then integrates the equations of motion and emits commands in the ANIM animation script language.

PM can deal with a variety of DT related simulations in a realistic fashion; for instance, it would have no trouble with the motion of a towed object such as a trailer or disabled vehicle. However, PM does not achieve realtime performance with any computing platforms available at UCF in 1991. In addition, the restoring forces model generates sets of DEs which are "stiff", requiring large numbers of small integration steps. More sophisticated mechanisms will be required before we can produce an acceptable realtime simulation of, say, the falling elements of a simple bridge after demolition.

PM is described in Appendix G.

3.4. Realtime Hydrology. In the summer of 1990, a paper by Miller and Kass appeared in the SigGRAPH Conference Proceedings which described a technique for the realistic realtime animation of flowing water. This paper inspired the production of a simulation on IST's Silicon Graphics workstations. Several phenomena were modeled, including

- conservation of volume during downhill flow
- accumulation of lakes and pools during rainfall
- Archimedean flotation and wave generation
- transport of floating objects on flowing water

The data gathered in these experiments was useful in testing various hypotheses about networked dynamic terrain simulations, as described in the following sections. The details of the realtime hydrology project may be found in Appendix H.

3.5 Soil Dynamics and Kinematics. The bulldozer described in section 2.2 above used only a crude model of the soil/blade interaction. Dr. John Farr of the Army Engineers Waterways Experiment Station in Vicksburg, MI provided a paper (Balovnev 63) with explicit models of dozer blade action. These nonlinear differential equations have not yet been incorporated into the dozer model, because of the need to solve database and networking problems prior to finely detailed physical modeling.

However, a simpler model for soil dynamics has been prototyped. This model assumes that a certain amount of soil slumping occurs in each time period, and redistributes soil via a relaxation algorithm. This helps to eliminate vertical edges of ditches and other artifacts which would not long survive the passage of a dozer. The algorithm is describe in Appendix I.

Also, a conceptual study was done which proposed to deal with detached bodies of soil such as those moved in a dump truck, or pushed by a dozer, as discrete objects, and to organize them on the basis of a grammatical, or syntax-structured paradigm. It was generally concluded that the resulting data structures would be needlessly complex for the benefit gained, and difficulties would arise in subsequently treating the ground as a homogenous whole. This study is provided as Appendix J.

4. Database Issues

4.1 Goals

In order to support dynamic terrain on distributed simulators, basic decisions must be made concerning the organization of the distributed database. Solutions could range from simple "ad-hoc" techniques which are really just patches to today's technology, to comprehensive reconsiderations of how visual databases are stored.

Because the present project was intended to develop basic technology, we avoided the most obvious short term solutions, which would involve the transmission of microterrain patches to image generators of today's architecture. This requires little more than engineering, and can probably be made to work for training tasks in which the dynamic terrain's ultimate configuration is all that matters.

For intimately interactive work such as the realtime construction of earthworks (with multiple bulldozers working in teams) or for most projected Virtual Environment applications (such as urban conflict, with munitions blowing holes in walls, etc.) a higher order of realtime control over the shared database is required.

In the previous phase of this project (1989), research was conducted into Object Oriented Databases as a possible support environment for Dynamic Terrain. Because we have continued to accumulate information about object technology during the 1990-91 project, we report here on the overall state of knowledge.

Section 4.3 then describes theoretical and experimental work on strategies for distributing shared data across multiple simulators.

4. 2. Object Oriented Databases (OODB)

The 1989 forerunner Project acquired the Gemstone OODB in order to design a prototypical object oriented terrain database. However, Gemstone proved a disappointment in that its typical response time to geographical queries exceeded 30 seconds. Even in a lab setting this was too slow to use for experimentation.

Using lessons learned in the construction of the Gemstone OODB, a second terrain database was constructed in Smalltalk. While still not a "realtime" system, this database was successful in serving as a testbed to build class hierarchies for terrain, develop editing concepts for regions and subclasses, and learn enough to include OODB concepts in future terrain database work.

In 1991 there has been a veritable explosion of progress in realtime object oriented software. Industrial users of OO languages are reporting that it is now possible to achieve excellent performance for certain time-critical tasks with specialized versions of Smalltalk. This will serve as a starting point for future VSL work on Dynamic Terrain.

The current state of the art in OO databases for Dynamic Terrain is reported in Appendix K.

4.3. Distributed Terrain Databases

One of the critical questions that must be answered for distributed simulation is how to handle redundant data. In SIMNET, all terrain data is redundant. That is, each simulator has a complete and identical copy of all the terrain data. However, such a strategy must be reexamined for use with dynamic terrain. The correct solution will vary tremendously, depending on the requirements.

Consider a single bulldozer, rearranging the soil. Its effects must be transmitted to all other simulators, even though most of them may never visit this location in their databases. If a sufficiently low update rate is acceptable, this can be done as a "background task". Latencies of up to several minutes may occur before the dozer simulator has produced a sufficiently simplified polygonal model of the disturbed terrain to send it as a microterrain packet to the other simulators.

If, however, two dozers are supposed to be working together in the classic "T maneuver", whereby one digs a ditch and the other crosses the T to remove the excess soil, both machines (and possibly several pairs of dozers) must have realtime views of the shared changing terrain. It makes more sense to move the soil and vehicle dynamics simulations to a common platform and to send the results to the various visualization sites. This could be done in several different ways.

One could use a single central soil processor, with sufficient power to handle the entire disburbed region and all dozers. One could use several processors, one per 'dozer, with each "owning" a patch of terrain centered on the vehicle. Or one could cut the terrain into patches with various degrees of granularity, assigning a processor to each patch.

Theoretical studies were conducted of various options along this spectrum, with predictions that the notion of granular dissection of the terrain database would prove most efficient, in the sense of minimizing the number of bytes per second which must cross the network.

The realtime experiments which were conducted were limited by the number of workstations available. At the level of 4 computing platforms, the overhead cost exceed the gains from parallelism, and so the hypothesis was not confirmed. It will be necessary to simulate or build a larger network of computing platforms before we can confirm the hypothesis for large 'n'.

Results of the analysis and experiments are provided in Appendix L.

5. Networking Issues

5.1 Goals

Experiences with the SIMNET system inspired the Distributed Interactive Simulation standards process (DIS 91), carried out by IST with Army sponsorship and industry-wide participation. A great deal of thought and effort was devoted to cleaning up and documenting the assumptions and conventions of packet format for future distributed interactive simulations.

Two components were missing, with respect to dynamic terrain:

- no standards were created for terrain database formats or behaviors, and
- no account was made of object oriented concepts such as class hierarchies, frameworks and protocols (in the data abstraction sense).

5.2 The Virtual Environment Realtime Network

In order to experiment with networked dynamic terrain simulation, the Project developed a family of internal prototypes of a communications architecture. This system used DIS concepts but couched them in an object oriented framework, and implemented the experiment on Unix workstations. The principal DIS concept used was dead reckoning.

Three major versions of the VERN (Virtual Environment Realtime Network) were produced:

- An architectural prototype in Smalltalk,
- A synchronous deterministic system, in C++,
- An asynchronous, high performance nondeterministic system in C++.

By "deterministic", we mean that all the simulation frameworks on all linked platforms were running in a common timing structure. Thus, from the same starting conditions, two simulations would always come out the same way. The synchronous version was built first in the expectation that it could serve as an analytical testbed for experiments in distributed physics.

In the asynchronous nondeterministic version, each local simulation ran as fast as possible, using system (realtime) clocks to control the integration process in the models. Consequently, because of the nature of Ethernet's collision mechanism and Unix timing, two runs will in general produce slightly different results. This is also the case with SIMNET.

With man-in-the-loop simulation, realtime responsiveness is essential, and small errors in integration are of less important. A tank driver doesn't know or care that his model would ideally have had him 100m further North than he is after an hour's drive; he will correct his course as a natural part of driving. Thus the asynchronous version of VERN is more likely to be used in future training simulation experiments.

The details of VERN's versions and history are provided in Appendix M.

6. Summary

6.1 Accomplishments

The Dynamic Terrain Project has explored the essential technologies for simulating and visualizing realtime dynamic terrain operations. These included

- visualization
- physical modeling
  - rigid bodies
  - soil manipulation
  - hydrology
- database organization
- communications protocols

In each of these areas, existing literature and software solutions were investigated; hypotheses were put forward; prototypes were constructed, and results were analyzed. We can now describe with some confidence the components of a working dynamic terrain simulation at some specified level of fidelity, and project the computing power required.

We did not construct an *integrated* set of demonstrations which incorporated all these components into a single system. The limitations of Unix workstations "playing the role of" realtime simulators proved to be too severe to make such a demonstration of practical value. However, the individual demonstrations of dynamic terrain components have proven very valuable for concept formation and for demonstrating feasibility.

A substantial amount of technology transfer into the new world of Virtual Environments has occurred, and will benefit the entire simulation industry in future years. Specific examples of this include the VERN system and the constraint-based physical modeling studies, which have led to two Ph.D dissertations and numerous Masters' theses and publications.

6.2 Recommendations for Future Work

Two diverging streams of work suggest themselves:

- further work on basic technologies for Dynamic Terrain, particularly the problems of database organization. Our simulation studies on novel partitioning strategies were inconclusive due to the small number of workstations employed.

- construction of a testbed leading to a deployable DT simulation. using a "strap-on" system which provides high fidelity realtime terrain viewing only to the crews who need it, and slower updates to the remaining DIS players.

Both of these avenues have been proposed to PM-TRADE as continuation work in 1992-94. A preponderance of the effort in the proposal is directed toward the testbed, as a component in the BDS-D Advanced Technology Transition Demonstration.

References

Balovnev, V. I. "New Methods for Calculating Resistance to Cutting of Soil". Translated from Russian; published by U. S. Dept. of Agriculture, Washington, D. C. 1963.

Moshell, J. Michael; C. E. Hughes, B. Goldiez, B. Blau, X. Li. "Dynamic Terrain in Networked Visual Simulators." VSL Document 89.17. December 1989

Appendix A:

List of Software Modules Produced by the Dynamic Terrain Project 1990-1991

File Name	Software	Author
VSL91.1	Polhemus Object	Richard Dunn-Roberts
VSL92.1	Constrained Dynamics	Julie Carrington
VSL91.3	Terrain Relaxation	Julie Carrington
VSL91.4	Terrain Editor	Li Xin
VSL91.5	Car Animation	Li Xin
VSL91.6	Bulldozer	Li Xin
VSL91.7	Beaver Pond	Chuck Campbell
VSL91.12	ANIM	Curt Lisle
VSL91.13	GIT/GMS/Multigen Formatter	Julie Carrington
VSL91.14	S1000 Display Utility	Bob Buckley
VSL91.15	Continuous LOD	Dan Mapes
VSL91.16	VERN 2.1	Chen Jinxiong
VSL91.17	Multigen to ESIG Formatter	Chen Jinxiong
VSL91.18	Database Experiments	Li Xin

Appendix B:

List of Publications Produced by the Dynamic Terrain Project 1990-1991

Blau, Brian; Charles E. Hughes, J. Michael Moshell, Curtis Lisle. "Networked Virtual Environments". *Computer Graphics 26:2,* March 1992. Special Issue on 1992 Symposium on Interactive Computer Graphics.

Burg, Jennifer; Charles E. Hughes, Michael Moshell. "Constraint-Based Modeling of Behaviors". *Proceedings, FLAIRS '90 (Florida Artificial Intelligence Research Symposium).* Cocoa Beach, FL. 3-6 April 1990.

Burg, Jennifer; Charles E. Hughes, J. Michael Moshell, Sheau-Dong Lang. "Constraint-Based Programming: A Survey". IST Technical Report IST-TR-90.16, August 1990.

Moshell, J. Michael. "Developments in Visual Simulation". *Military Simulation and Training,* 5/91. Monch Publishing Group, Farnborough, Hants., UK.

Moshell, J. Michael, Charles E. Hughes, Brian Blau, Xin Li, Richard Dunn-Roberts. "Networked Virtual Environments for Simulation and Training". *in Proceedings, SimTech '92 (Society for Computer Simulation), Orlando, FL. 21-23 Oct. 1991*

Moshell, J. Michael, Charles E. Hughes, Brian Blau. "Networked Virtual Environments: Issues and Approaches." *in Proceedings of the SRI Virtual Worlds Symposium,* Menlo Park, CA. June 17-19, 1991.

Moshell, J. Michael, Ernest A. Smart, Richard Dunn-Roberts, Brian Blau, Curtis R. Lisle. "Virtual Reality: Its Potential Impact on Embedded Training".*Proceedings of the NATO RSG16 Workshop on Advanced Technologies Applied to Training Design.* Venice, Italy. October 22-24, 1991.

Moshell, J. Michael; Xin Li, Charles E. Campbell, Curtis R. Lisle. "Dynamic Terrain Databases on Networked Visual Simulators". *Proceedings of the Image VI Conference* , Phoenix, AX. 14-17 July 1992. Image Society of America.

Moshell, J. Michael, Xin Li, Charles E. Hughes, Brian Blau, Brian Goldiez. "Nap-of-Earth FLight and the Realtime Simulation of Dynamic Terrain." *Proceedings of the SPIE Conference.* Orlando, FL. 16-20 April 1990.

Appendix C:

Terrain Profile Experiments:
The Virtual Bulldozer

Dynamic Terrain Project
1990-91

Terrain Editor

Program description

The program demonstrates the idea of using Cardinal splines to represent terrain. A piece of terrain is stored in a two dimensional array in the program, and displayed in the three dimensional space on the screen. Using different combinations of mouse buttons, a new piece of terrain can be created, modified by adding hills and valleys on it, or saved in a disk file. The program also provides a three dimensional view of terrain. The image can be zoomed, shaded and rotated.

To run the program:

1) type "terrain < imagefile name>" to run the demo. If the file does not exist, a new terrain file is created.

2) Mouse buttons:
left button -- show the red crosshair cursor;
middle button -- rotate the terrain image*;
right button -- shade the terrain image;
left & middle buttons -- pull up a hill where the red crosshair is;
left & right buttons -- push down a valley where the red crosshair is;
middle & right buttons -- rotate the shaded terrain image*;

* The direction of rotation is controlled by the mouse position in the window.

3) Keys:
"z" key -- amplify the terrain image;
"x" key -- shrink the terrain image;
"w" key -- save the terrain image;
ESC key -- quit;

Source Code files

Project #: VSL91.6
Directory: /sg1/files/home/student/lix/demos/bulldozer
source file: data.h terrain.c mkterrain.c shading.c sub.c

Car Animation

Program description

The program simulates a vehicle driving on a piece of terrain. A piece of terrain is stored in a two dimensional array in the program, and displayed in the three dimensional space on the screen. By different mouse buttons, the vehicle can be turned right or left, driven forwards, backwards, or stopped. The velocity of the vehicle can also be changed by pressing keys. Simple physical properties of moving vehicles (i.e. the gravity, the friction and the air resistance) are also modelled in the program. The acceleration is gained by hitting "a". The friction is proportional to the velocity of the vehicle, and the air resistance is proportional to the square of the velocity.

To run the program:

1) type "motion < imagefile name>" to run the demo. If the file does not exist, a flat terrain is used.

2) Mouse buttons:
left button -- turn the vehicle left;
right button -- turn the vehicle right;
middle button -- drive, reverse or stop the vehicle;

3) Keys:
"a" key -- increase the speed;
"s" key -- decrease the speed;
ESC key -- quit;

Source Code files

Project #: VSL91.5
Directory: /sg1/files/home/student/lix/demos/car
source file: data.h func.h main.c vehicle.c car.c mkterrain.c shading.c sub.c

Bulldozer

Program description

The program combines the ideas of the terrain editor and the car animation to simulate a bulldozer driving on and modifying a piece of terrain. By different mouse buttons, the bulldozer can be turned right or left, driven forwards, backwards, or stopped. Its blade can be raised and lowered by pressing keys. When the bulldozer is driving forwards with its blade down, it digs a trench and piles dirt to the front and sides of the blade. The model is kinematic, so the treads will not slip, dirt will not spill back down into the trench, and the bulldozer is not limited in the amount of earth it can move. The volume of soil is not conserved, although it looks like it is.

To run the program:

1) type "motion < imagefile name>" to run the demo. If the file does not exist, a new terrain file is created.

2) Mouse buttons:
left button -- turn the vehicle left;
right button -- turn the vehicle right;
middle button -- drive, reverse or stop the vehicle;

3) Keys:
"a" key -- increase the speed;
"s" key -- decrease the speed;
"u" key -- raise up the blade;
"d" key -- lower down the blade;
"r" key -- set the blade to the normal position;
"w" key -- save the terrain image;
ESC key -- quit;

Source Code files

Project #: VSL91.6
Directory: /sg1/files/home/student/lix/demos/bulldozer
source file: data.h func.h motion.c vehicle.c bulldozer.c mkterrain.c shading.c sub.c

SIMNET Bulldozer

Program Description

The program combines the ideas of the bulldozer simulation with realistic SIMNET terrain. The 125 meter basic terrain square polygons are broken into two triangles. When the dozer enters one of these triangles, it is cut into 5 meter triangles, forming microterrain. No terrain relaxation is performed, and so this algorithm would be impractical in a real simulator, but using only wireframe rendering its speed is acceptable for demos. Depending on the workstation the simulation runs at 1 to 5 frames/second, until a substantial number of 125 m polygons have been broken up.

To run the program:

1) type "motion < imagefile name>" to run the demo. If the image file does not exist, a flat terrain is used.

2) Mouse buttons:

left button -- turn the vehicle left
right button -- turn the vehicle right
middle button -- drive, reverse or stop the vehicle

3) Keys:

“a” key - increase the speed (actually the step size)
“s” key -- decrease the speed
“u” key -- raise the blade
“d” key -- lower the blade
“r” key -- set the blade to neutral position
“w” key -- save the terrain image

Source Code Files:

<Micheline or Henry to provide>

Appendix D:
Visualization Studies

Dynamic Terrain Project
1990-91

The Visualization of Dynamic Terrain

J. Michael Moshell
Curtis R. Lisle

Institute for Simulation and Training

VSL Document 91.20

10 December 1991

Abstract

Realtime networked graphical simulators such as SIMNET [Johnson 87] are of intense current interest, both for military training and for the larger market of commercial, entertainment and educational applications currently being called "Virtual Reality" [Furness 88]. However, no existing realtime simulation supports a truly interactive world. In particular, the terrain (soil, water and vegetation) is nearly or completely immutable in today's simulators. In a word, the terrain is not dynamic.

This paper contains an analysis of the cost of visually displaying dynamic terrain at a level of fidelity adequate for many military training purposes, and for Virtual Reality as it is likely to evolve in the 1990's. The display of dynamic terrain seems to be generally within the capabilities of moderate-cost visual systems that can be built by the mid-90's. Exceptions are noted as they arise in the text.

This paper addresses only two of the problems raised by dynamic terrain: the polygon and texture requirements for visualization. Other papers from this project will address the issues of realtime physical simulation (e. g. of hydrology), database representation and network protocols for dynamic terrain.

This work was sponsored by the U. S. Army's Project Manager for Training Devices (PM-TRADE). However, all opinions are solely those of the authors.

Introduction

Dynamic Terrain (DT) denotes the capability, within a realtime graphical simulation, of rearranging the terrain surface. DT essentially involves allowing the simulation's user to dig holes in the terrain database at any freely chosen location, with the expectation that the simulated world will behave appropriately in response. Other broadly distributed unplanned modifiable features such as vehicle track marks, vegetation damage and hydrology are often included in discussions of DT, since they require similar data structures and are often the consequence of realtime digging.

This paper contains an analysis of the cost of visually displaying dynamic terrain at a level of fidelity adequate for combined arms tactical training and combat development. We treat only briefly the related issues of how the changes to terrain might be shared among networked simulators, and will refer to other papers addressing these problems.

The broad conclusions of our study are:

a. An acceptable display for engineer, infantry, armor and artillary purposes is achievable by the mid-90's on simulators having approximately twice the graphical performance of current SIMNET units; but their visual systems will be substantially different from those in SIMNET.

b. Aircraft simulators of SIMNET/AirNet quality, already regarded as unacceptably crude by much of the aviation community, will be unable to render dynamic terrain in other than a token fashion. Training utility of such a system will have to be assessed.

c. The minimal DT display system for a ground-based viewpoint will require an image generator capable of displaying 2000 to 4000 polygons at 15 frames/second per channel, equipped with appropriate database access and control algorithms in the front-end geometry processor. The number of required channels is determined by the vehicle being simulated.

CONTENTS

1. Goals of the Study

No clear and comprehensive statement of the technical requirements for Dynamic Terrain (DT) in distributed interactive simulation (DIS) are known to the authors. Our working informal list of desirable features is described below.

NOTE: Not all these features are achievable with today's technology. The limiting factors are described in the subsequent text.

1. SOIL.

a. At every location on the terrain database, attributes describing the vegetation cover, soil type, condition and moisture content should be recoverable. These attributes should be modifiable by simulated events (e.g. traffic, precipitation) during a simulation, when appropriate. A specific example is that tread marks of passing vehicles should appear if soil conditions would permit.

b. The visual appearance of the terrain (its surface texture) should change to reflect the current values of the attributes described above. For instance, wet soil should usually appear darker than dry soil of the same type.

c. The terrain elevation profile should be modifiable to represent the effects of earthworks construction and breaching, explosions and weathering. The modified elevation profile becomes “ground truth” for all simulation elements, and can be used as concealment, protective structures, roadways etc.

d. Soil should slump to its angle of repose, according to its specific characteristics (sand, gravel, mud, stone, etc).

2. WATER.

a. Water flowing across the terrain or standing in low places should be represented in a visually realistic fashion.

b. Bodies of water should behave realistically; seeking the lowest level, being absorbed into soils of certain types, evaporating, carrying floating objects along.

c. Precipitation should be supported, and should result in the development of drainage patterns as required by the topography.

3. OBJECTS.

Discrete rigid objects such as boulders, logs and reinforced concrete bridge elements should be representable in a physically realistic fashion; i. e.

a. Objects should obey Newtonian physics. When unsupported, they should fall.

b. Objects should be capable of being pushed by a simulated bulldozer or other prime mover, if not too massive, and leave drag-marks on the ground. A pushed object should also be able to push other objects it encounters.

c. Objects should break into pieces when attacked with appropriate explosives or ordnance. The resulting pieces should obey a) and b) above.

2. Aggregated Features and Achievable Densities

The display of soil, water and objects in a visual simulator is primarily limited by the number of polygons that can be rendered per channel, per frame. As a starting point, we will evaluate the polygon budget of one channel of a hypothetical image generator named MARK 1 vis-a-vis some simple DT features.

The MARK 1's overall capabilities closely mirror those of the basic SIMNET M1A1 tank simulator system, while not exactly matching any of the several SIMNET versions in existence. We will then explore the consequences of doubling or quadrupling the MARK 1's capacity to produce MARK 2.

DT Features. Let us start with the desired results, and work backward to the means of achieving them. A short list of interesting and desirable local features would include craters, emplacements (short ditches, with or without berms) and track marks. Other obstacles such as tank ditches can be considered once some basic ideas are established.

> By "aggregated features" we mean features constructed from the smallest number of polygons, rather than more realistic micro-terrain (typically 1 sq m patches) produced by the DT bulldozer. The following section discusses the relationship between microterrain and aggregated features.

The following sketches show that "cheap" and "nice" (irregular) versions of emplacements and craters can be built with between 10 and 50 polygons per feature. These features will of necessity be somewhat cartoon-like. Their purpose in DIS is to obstruct and conceal, not to edify.

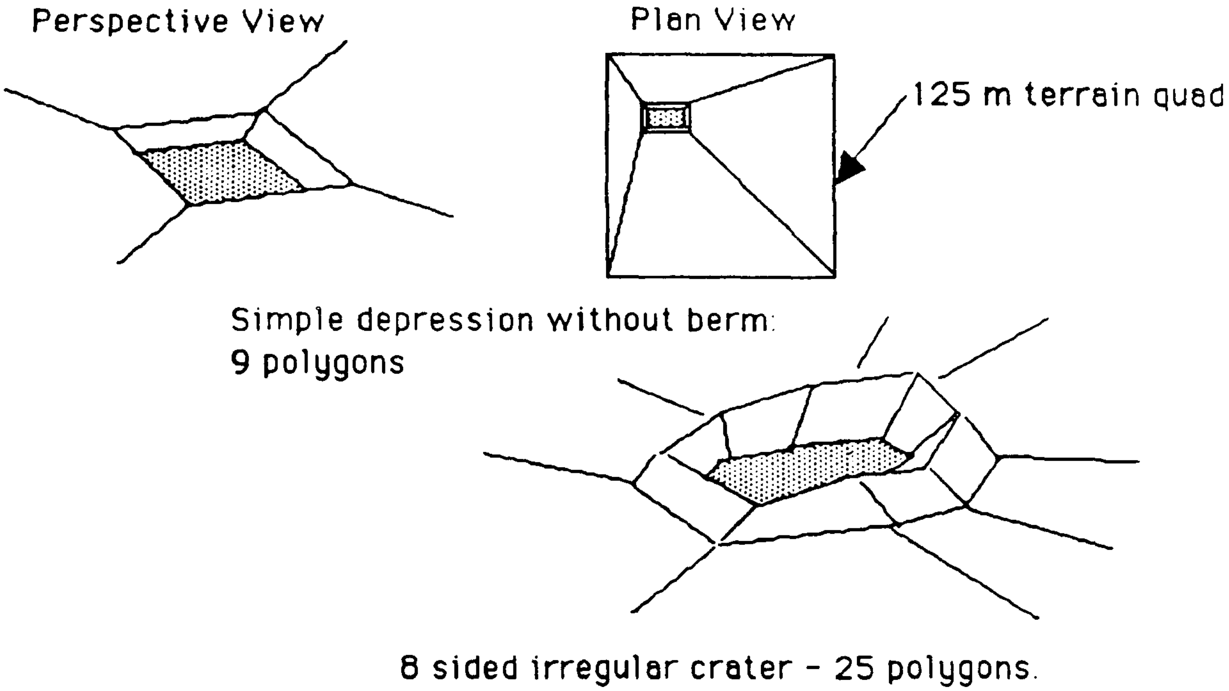

Perspective View
Plan View
125 m terrain quad
Simple depression without berm:
9 polygons
8 sided irregular crater - 25 polygons.

Figure 1: Simple Polygonal Depressions and Craters

Tread marks will be considered in two forms: straight line travel, and maneuvering. Straight (or essentially straight) travel results in relatively long segments that can be represented by a single pair of textured polygons. Maneuver obviously requires more polygons, as the tank changes course. Polygon counting will require some understanding of MARK 1 architecture.

Load Modules and Terrain. MARK 1 channels provide 1000 polygons per frame, at 15 frames/sec. In fact, some spare capacity is afforded by the fact that MARK 1's channels are paired, and that one of the channels (the "primary") can display in excess of 1000 polygons, with the consequence that other channel's scene is correspondingly degraded. This capability is best used as a margin for error, in case many movable models crowd together or some complex static features are inadvertantly sited close together during database construction.

The basic strategy for MARK 1 database construction is to use 300 of these polygons for terrain, 400 for static models ("culture") and 400 for moving models. Thus, the database cannot be so densely built that more than 300 terrain polygons are visible from any accessible viewpoint and viewing direction, or (when culture and moving models are present' the channel's capacity will be exceeded.

A standard view window is a 20 degree wide by 7.5 degree high field of view, and looks out onto a Local Area of nominal radius 3.5 km. The visible terrain (resident in the Local Area Memory of the IG) consists of an array of 16 x 16 Load Modules of 0.5 sq km each.

One Load Module (LM) normally consists of a 4 x 4 regular array of polygons, which may be occupied by squares or triangles (depending on the irregularity of the surface). There are at most 32 of these 125 m polygons in a LM. Coplanar polygons may be combined in a process called terrain relaxation, and so there may be as few as 1 polygon in a LM.

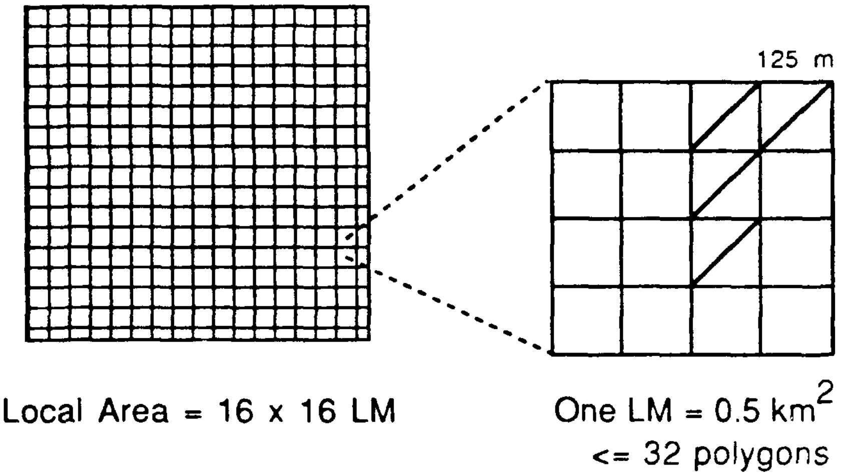


Figure 2: Load Modules

Since the viewpoint is always in the approximate center of the Local Area, the cone of vision subtends an area approximately equivalent to 9 Load Modules or at most 288 terrain polygons. This fits, barely, within the 300 polygon limit.

The Next Generation. Let us imagine an improved image generator, in which 600, rather than 300 polygons were available for terrain. We now have 300 additional polygons available for dynamic terrain features. How densely would this allow us to populate the terrain with tread marks, craters, and emplacements?

In a MARK 1 viewing channel there are 128 distinct horizontal lines of imagery. This low vertical resolution is justfied by the fact that the viewing blocks (periscopes) of the M1A1 tank are rather thin horizontal slits. At a range of 1 km, a 2 m high berm or crater wall subtends 0.11 degrees, and is two pixels high; essentially indistinguishable.

> We might agree to not display dynamic terrain features at ranges greater than 1 km. However, if a greater vertical resolution were used, we might instead have to use level-of-detail control, and cause remote objects to transition to simpler polygonal models. This subject will be discussed in Section 4 below.

Note: Vehicles can still get "hull down" in a DT emplacement or obstacle, at whatever range from the viewer. The vehicle will sense the emplacement's bottom and get its elevation from it. From a range of >1 km, the vehicle will seem to be sinking into the larger terrain polygon (because the DT feature will be invisible). If the hole is deep enough the vehicle will disappear, as it would in a fully rendered hole.

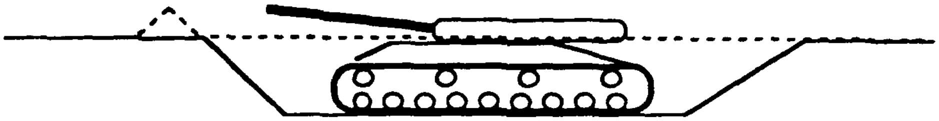
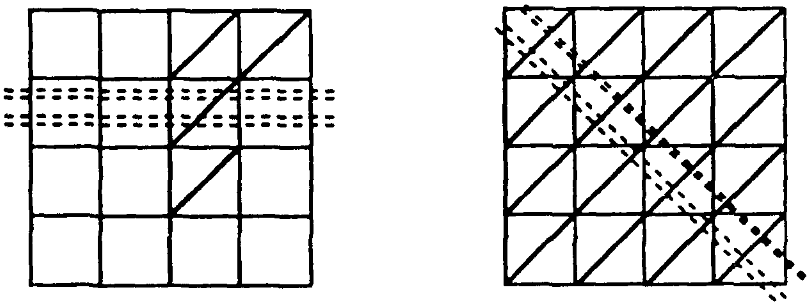


Figure 3: Tank in Defilade

Under this 1 km range limiting assumption, the 20 degree viewing cone now contains only 0.18 square KM of dynamic terrain.The available 300 polygons thus corresponds to a DT polygon density of 300*(1/.18) = 1667 polygons per square km.

Tread Marks. We assume that the MARK 1 IG is capable of prioritizing polygons, so that coplanar polygons do not confuse the depth buffer. Tread marks will be represented by a sequence of rectangular textured polygons, representing the disturbed surface of the soil.

Traveling in straight lines, a tank would cross between 1 and 8 terrain polygons in the "best case" (parallel to the grid and within one 125 m strip of polygons), generating 2 to 16 track-mark polygons. The worst case ("against the grain", diagonally across a LM) would generate 28 track-mark polygons per load module.



Best Case: 2-16 polys

Worst Case: 28 polys

Figure 4: Tracks across load modules.

If the tank maneuvers once per five tank-lengths (40 m), or (say) 12 maneuvers per half kilometer (one Load Module), and if half of the 40-meter travel segments crossed between two of the 125-m quads, then we would generate on the average 12*(1.5)*2 = 36 polygons per load module.

Thus, the practicaı range of polygon production is perhaps 12 to 36 polygons per load module, or 24 to 72 polygons per km of linear travel. If tank tracks were the only DT feature imposed on terrain, between 23 and 69 tanks couid cross through a 1 sq km area before the available polygon count per km (1667) was exceeded.

When this occurs (or when polygons are needed for other DT features) a garbage collection procedure will be necessary to reclaim the oldest tread mark polygons.

Craters and Emplacements. Craters and emplacements need both a berm or spoil bank above the grade, and a bottom below grade. Minimal triangular craters can be drawn with eleven polygons in place of the original quadrilateral; more realistically irregular craters would require 15 to 50 polygons. A simple square emplacement without a berm consumes nine polygons. (See Figure 1 above).

The permissible density of craters and emplacements, again with 1667 polygons/sq km, is now between 33 and 128 per sq km. This value is almost independent of the actual size of the craters or emplacements, as long as the features are substantially smaller than the 125 m terrain polygons.

Anti-Tank Ditches. A ditch with berm(s) can be considered as a series of emplacements, for polygon counting purposes. Each straight segment of ditch of length less than 125 meters counts as one emplacement. Thus, an anti-tank ditch which zig-zags every 30 m from left to right across the field of view at a range of 0.5 km would be equivalent to six emplacements or 78 polygons.

Significance. Now it may be remarked that 33 to 128 craters per sq km is a very sparse population. Indeed, they must remain somewhat dispersed because we have only assumed 300 new DT polygons, and so at most six 50-polygon craters could fall within the field of view without overloading the IG.

Two responses suggest themselves.

- First, that's six more DT features than we have now in SIMNET.

- Second, we can actually expect not an increase of 300, but 1000 or more new polygons per channel. Computer graphics capabilities have continued to double for a given price every two to three years since SIMNET was produced.

- Third, we have not yet proposed multiple levels of detail. If we allow high fidelity (50-polygon) craters to be replaced by crude (20-polygon) craters at 0.5 km range, another doubling in capacity is possible. Tread marks are likely to become invisible at much less than 0.5 km becaue, they lack vertical relief.

Thus, we can have six complex craters AND 20 emplacements AND the tread marks of around 60 tanks in the field of view at once. Or we can spend our entire budget on anti-tank ditches, and have five or more zigzag ditches between the viewpoint and the 1 km "complexity horizon". Numerous scenarios suggest themselves.

Some Samples. The following three pictures show some typical polygon densities. A more comprehensive set of pictures is found in a separate report [Moshell 91]. First, we show a simple terrain scene, rendered at what we might call MARK 2 resolution - that is, a 9 by 40 degree field of view in 250 x 930 pixels. This yields an average angular pixel density of 2.25 arc-minutes per pixel, which according to [Bess 90] is an appropriate density for a tank commander's unmagnified viewing block. The scene contains about 330 terrain polygons and four tanks, of 226 polygons each. Only the nearest tank needs to be rendered at high level of detail.

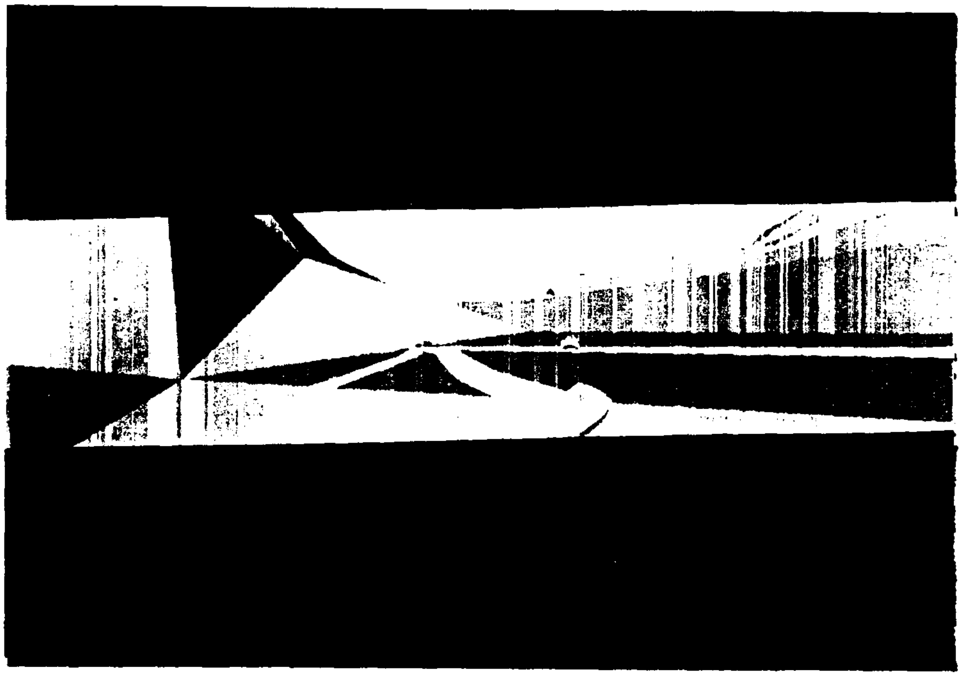


Figure 5: Background Terrain, 330 polygons.

Now let us add some craters, berms and tread marks. Specifically, we add the following features to form Figure 6:

Feature	**Count**	**Polys/feature**	**Total Polys**
Berm	6	12	72
Crater	6	12	72
Track	2	56	112
Total Dyn. Terrain			256

The resulting scene is shown in Figure 6. The count of polygons in the tracks includes "generated" polygons which were necessary when the tracks crossed from one terrain poly to another, but invisible in the scene.

The six craters are difficult to locate since their range from the viewer varies from a few tens of meters to 1/2 km.

A triangular tree, 25m tall, was placed in the upper central portion of the scene at a range of 1 km to aid the viewer in locating the most remote feature cluster, which is a tank parked behind a berm. The yellow dot (left and down from the tree) is a muzzle flash from the tank. Even though this scene has over twice the vertical resolution of the MARK 1 system, it seems clear that a non-moving tank is nearly undetectible and definitely unidentifiable at that range.

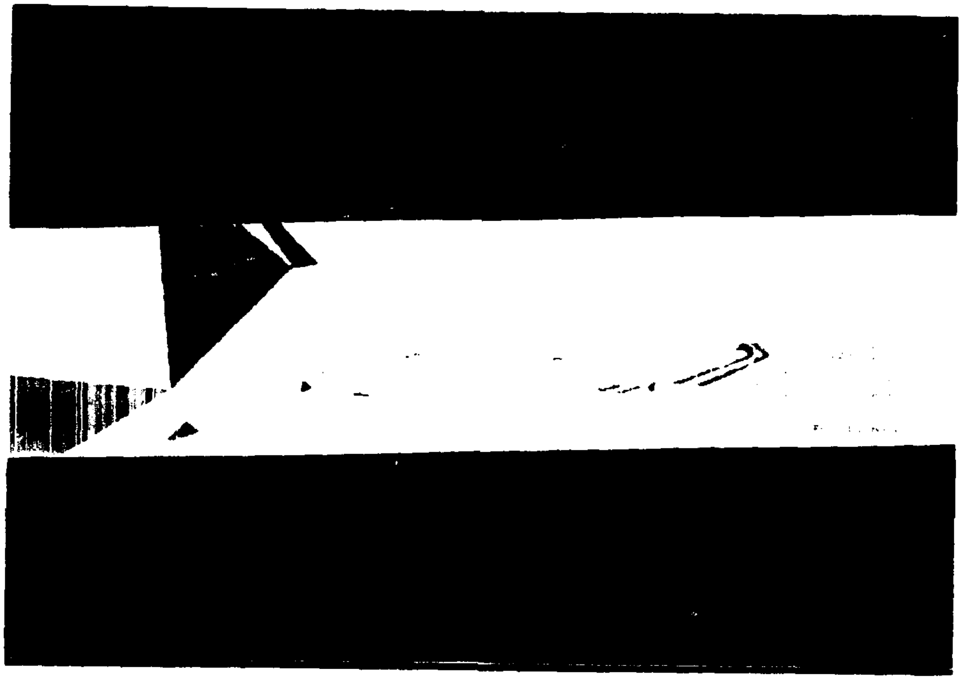


Figure 6: Cheap Dynamic Terrain, 256 DT Polygons

To explore a higher-fidelity solution in Figure 7, we now quadruple the polygon count to approximately 1000 polygons for dynamic terrain features, as summerized in the following table.

Feature	Count	Polys/feature	Total Polys
Berm	10	56	560
Crater	6	48	288
Track	2	101	202
Total Dyn. Terrain			1050

In this scene, we see that the tread marks of the nearest tank bend more often than in Figure 6 and that the berms no longer have an angular appearance. The difference in the appearance of the craters is almost unnoticeable.

The on-screen display of these images, on the Silicon Graphics computer, is somewhat crisper than these photographs. In Figure 7 as seen on screen, a smaller berm is visible to the left of the remote (1 km) berm concealing a tank. According to Johnson's Criteria [Biberman 73], the remote tank lies between the maximum feasible detection range (3300m) for this pixel resolution and the reliable recognition range (710m).

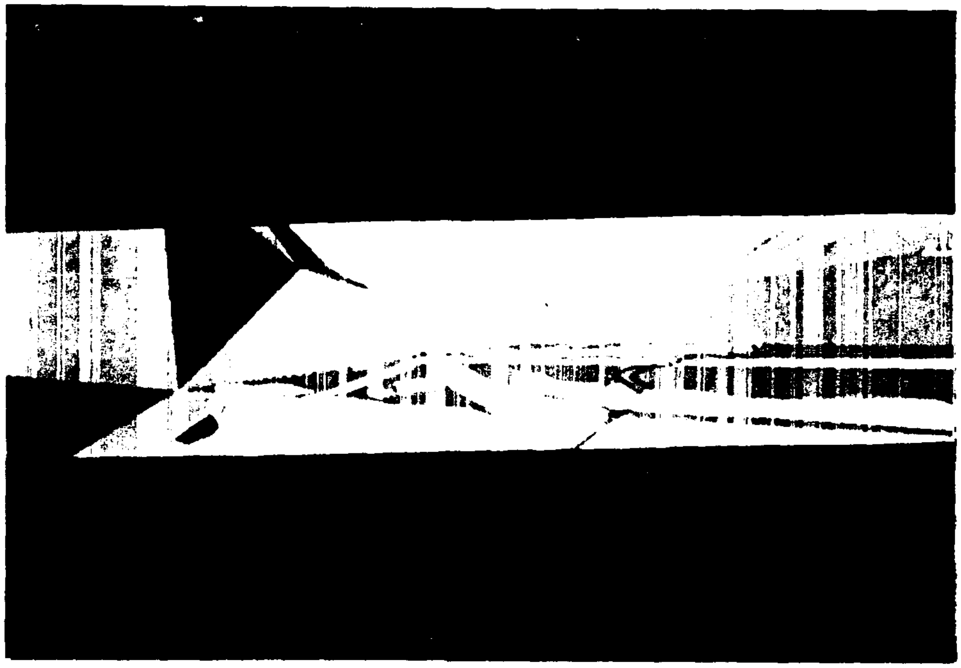


Figure 7: Better Dynamic Terrain, 1050 DT Polygons

The Problem of Aviation. Aircraft can see far more than 0.18 square km of terrain, of course. Let us examine the polygon load on a SIMNET viewpoint in an elevated location. In order to conservatively generate a lower bound on the polygon count, we will continue to assume only a 7.5 by 20 degree field of view. Any reasonable aircraft simulator would need a much larger FOV per channel.

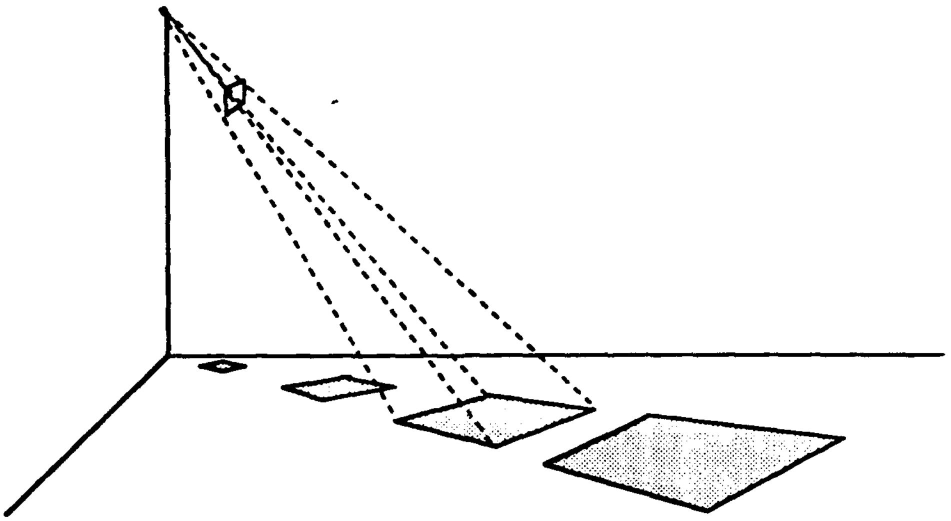


Figure 8: Different Lookdown Angles and Ground Coverage

At 1 km altitude, a 7.5 x 20 degree field of view sees the following area of level terrain, at various look-down angles as computed using the perspective projection:

View Angle	Area of Terrain
0°, straight down	.05 sq km
20° from vertical	.06 sq km
40° from vertical	.12 sq km
50° from vertical	.22 sq km
60° from vertical	.55 sq km
65° from vertical	1.03 sq km; far edge at 2.9 km range
70° from vertical	2.34 sq km; range is 2.7 to 4.5 km
75° from vertical	7.85 sq km; range is 3.7 to 7.2 km

It is apparent that as the view direction approaches the 3.5 km "horizon", or range of the load module, the viewing area exceeds 1 sq km. Viewing over 1 sq km of terrain at once is a five-fold increase in polygon count over our previous ground-based and range-limited assumptions. In addition, the basic terrain skin itself now requires a five-fold increase in polygons.

Our assumption allowing DT features to become invisible at 1 km range is also suspect now, because we're looking down on them. The vertical angle subtended is proportional to the front-to-back depth of the feature, not just to its height.

The saving grace is that we are at a minimum of 1 km range from any static or moving models and can use coarser levels of detail to recover polygons from them. Users of aviation simulators also favor rapid update rates (e. g. 30 hz and up) rather than (or in addition to) high polygon density, and so the entire design process tends to diverge from that for ground based simualators.

Our tentative conclusion is that the provision of DT for aviation simulators will require subterfuge, and will be less visually convincing than for ground applications. A single "footprint" semi-transparent polygon containing a texture map appropriate to disturbed terrain, for example, might be substituted for a 50-polygon emplacement.

This would provide some degree of cover and concealment if the underlying elevation profile allowed a tank to bury itself in the textured polygon, and yet would allow the tank to be partially visible when the aircraft passed overhead. This cover-up solution would be less acceptable for partially defiladed vehicles; their profiles would be totally exposed to aviation.

In terms of damage assessment after air strikes, a texture map of damage on a runway surface is probably an adequate cue.

Clearly some experimentation will be required to make the best of a strained situation. The authors do not assert that fully satisfactory aviation-viewed DT can be provided within the cost domain of current simulation projects such as the Close Combat Tactical Trainer (CCTT), a SIMNET follow-on for the mid-90's.

Back on the Ground. Even when limited to grounded viewpoints, there are other demands for increased polygon count. One obvious need is for broader fields of view. For instance a six-monitor simulation of tank vision blocks would require at least 40 degrees of width in the FOV's, and possibly as much as 60 degrees. This implies a doubling or tripling of polygon count, compared to SIMNET.

Let us now consider one major competitor for polygons within the dynamic terrain arena: micro-terrain - the immediate output of the bulldozer.

3. Micro-Terrain

Consider the cost of the profligate use of micro-terrain. Again using a 20 degree wide ground-based field of view as a conservative estimation tool, we discover that if terrain is dissected into 1 meter quadrangles, that within a range of 42 meters, the viewing cone already contains over 300 polygons. Looking all the way to the edge of the current 125 m grid sector can include as many as 2845 square meters.

The only possible conclusion is that the use of micro-terrain must be severely limited. In a special purpose simulator such as a bulldozer, one can imagine the use of one or a few visual channels with 3000 polygons, almost entirely devoted to terrain; but clearly this cannot be done for all of the networked simulators. Other priorities (such as the rendering of moving enemy vehicles) must also be met.

It will be necessary to develop techniques which use micro-terrain only at the precise points and times where soil is actively changing shape; and to combine this terrain into aggregated features as quickly as possible. The following section describes algorithms that may prove useful to perform this task.

Finally, it should be remarked that most viewing platforms do not need to see micro-terrain at all. Only the simulators actually used by engineers, or perhaps some of the combat elements watching engineers working under fire, need to follow the second-by-second development of the shape of a ditch or berm.

For the USERS of dynamic terrain (e. g. the combat elements which will follow the engineers through a breach) it suffices to see and interact with the resulting aggregated features. Let us now consider how these might be produced on the basis of primary microterrain.

4. Realtime Terrain Relaxation

How can micro-terrain be transformed into aggregated terrain in a realtime fashion, without producing unacceptable visual anomalies? We regard the instantaneous substitution of a simple, smooth crater wall for a lumpy piece of micro-terrain as unsatisfactory. Let us look for some techniques that might help.

Continuous Levels of Terrain Detail. In most image generators, the polygon burden of moving and static models is managed by maintaining multiple data structures for the same object, called "levels of detail (LOD)". At a given viewing range, transparency fading is used to replace a complex model by a simpler one, thus conserving polygons.

LOD for terrain could be handled similarly, but some problems arise. A transition between two terrain LOD's can modify the apparent elevation of a ridge line, for instance, revealing an object behind the ridge. Several studies (Chen 87, Ferguson 90, Scarlatos 90) have explored methods which minimize the changes to significant features (ridge lines and valleys) and which match edges and vertices between LODs.

If a "raw" terrain database can be processed, either beforehand or at runtime, to produce two or more LODs with these properties, then a realtime transition is possible. At a certain range, the image generator simply begins to draw polygons from the less dense LOD database.

If the elevation information is interpolated for polygons on the fringe of the transition area, the displayed polygons seem to rise or fall slightly as the viewpoint approaches; but the transition is usually done at such a range that the movement is invisible. Both General Electric and Evans and Sutherland now offer products incorporating these concepts.

Because all LODs are designed in a coherent edge-matched fashion, the transition does not produce gaps in the viewed image. The production of a hierarchical database appropriate for continuous LOD display is usually considered as an off-line operation. However, Evans and Sutherland assert that their ESIG-4000 can perform this task at runtime.

Terrain Relaxation. This term normally refers to the process of seeking, in a large polygonal terrain database, polygons of sufficiently similar slope and elevation that they may be merged into a single polygon. However, we may generalize the term to refer to the general problem of re-polygonizing terrain to preserve its essential shape while lowering the poly count.

The problem of transforming microterrain into aggregated features is clearly similar to the problem of producing coarse LODS from fine detail without seriously modifying the significant attributes, such as ridgelines. Would it be possible to design a realtime algorithm which transforms micro-terrain into efficient aggregated features? We believe so, for the following reasons:

- We know which polygons were recently disturbed. Only they and their adjacent polygons need to be considered, not the entire database.

- Only those polygons at or beyond the transition range (40 m in the above analysis) and still visible in the viewing direction, are urgently in need of relaxation. Disturbed polygons left behind by the dozer can be dealt with opportunistically, at a lower priority.

- CCTT and subsequent systems will use textured polygons for all terrain. Texture obscures fine detail, and thus the relaxation of the terrain in front of the immediate ridgeline/horizon is likely to either be unobtrusive, or to resemble normal slumping of the soil.

Experiments in this direction will be carried out during the next phase of research into Dynamic Terrain at the Institute for Simulation and Training. We are confident that micro-terrain can be smoothly combined with aggregated features in realtime DT displays.

5. Networking and Distributed Databases

Separate papers will address the basic problems raised by dynamic terrain databases for the networking of simulators, and will describe theoretical and experimental studies which we have conducted. Here we wish to touch briefly on an interesting question: just how intrinisically difficult will it be to propagate the results of a dynamic terrain operation across a DIS system? We will consider both soil and water.

Soil. As a first data point, we have measured the performance of our Virtual Bulldozer operating on one-meter polygonal terrain. The bulldozer modifies terrain elevation values in two fashions:

- by "clipping" a post when the blade crosses it, and
- by spillover, when the earth's volume is redistributed.

The dozer blade clips two to five elevation posts per simulation frame. The amount of spillover is dependent on the terrain algorithm used; our present spline-based algorithm (approximately volume-conserving) results in an average of 21 elevation value changes per frame (including the clipping), when the bulldozer is run through 1000 frames of typical earthmoving activity.

Now if remote simulators were maintaining micro-terrain, only the dozer's position actually need be transmitted through the net; the local computers could recompute the micro-terrain profile by repeating the original spline computation.

However, if increased realism (soil moisture and classification, physical blade models) were used, special computing power is more likely to be needed for the dozer simulator. Under these circumstances it makes more sense to transmit the terrain profile, than to re-compute the physical model at all sites.

Most combat elements don't need to know about micro-terrain. However, a pair of dozers working together would need to share the fine structure of the terrain and would thus have to exchange information at about the rate indicated here. If we assume an acceptable worst case of 30 changed elevation posts per frame and transmit only elevation changes (e.g. in cm), a DIS packet with around 30 data bytes would suffice to express 30 changes of up to ± 1.28 m per frame. This is the same order of magnitude as the traffic in vehicle appearance packets generated by a SIMNET tank.

Water. Soil has one desirable property: it doesn't move much unless you push it. Water, however, may be changing its height at every elevation post in an entire database, at every simulation frame. This is a taxing situation, both for the physical simulation and for the networking.

IST's realtime hydrology is based on the simplified hydrodynamics of [Kass 90]. The simulation CPU must examine every elevation post, alternating scans in a N-S and E-W direction through the body of water. We have demonstrated that this is possible for a local region (e.g. a 50 m diameter lake or a 100 m segment of a stream) using graphics workstations such as the Silicon Graphics 4D/240, achieving frame rates in excess of 5/second. With optimizations and the rapid decrease in the cost of powerful RISC CPU's, 15 hz operation is probably possible in our price range.

However, what about water that is currently not being viewed? If a dam is broken and the simulator (with viewing system) then moves elsewhere, when you return five minutes later, where is the water? On flat, featureless terrain the water would be everywhere.

Our working assumption, in order to produce useful hydrology within our time frame, is that water will be confined to finite channels whose total surface area does not exceed some manageable limit (e.g. 20,000 sq m). Several cheap and fast simulation CPUs can then share the task of maintaining this elevation profile.

Dozers and explosives will be able to breach dams, etc. but the water which runs out will not flow forever. Furthermore, the degree of aggregation (cell size) of the water in varous regions may require manual tailoring, in view of the training or simulation scenario to be exercised. Distant water doesn't need one-meter cells of elevation profiling.

With these limiting assumptions, how can water be propagated within DIS? One possiblity is to follow the bulldozer paradigm. Only a principal hydro simulator (perhaps not even a visually equipped one) manages actual hydrology, cell by cell. If, for instance, a float bridge is traversing a stream, the detailed hydro model with Archimedes principle would be executed by the simulator managing the bridge.

Other simulators will simply display fairly large flat polygons of the appropriate color to report the water surface's height, with the bridge elements imbedded in them like raisins in toast. The dynamic system will inform all simulators of the appropriate height of these "macro water skins", based on physical events such as a tank's crossing the bridge (which might displace water and raise the waterline along the stream and along the bridge elements).

These aggregated water skin polys are a hydro equivalent to the aggregated land features, except that instead of being left behind as the microterrain process goes elsewhere, these are repeatedly adjusted as the fine-grained model of the water dictates, to represent sloshing, damming, etc. Because the count of macro polygons is much less than the number of microsurface water cells, broadcast network traffic will again be tolerable.

6. Dynamic Terrain in Virtual Reality

Substantial interest in DT has been expressed to the authors by some commercial developers of realtime moving-viewpoint "flyover" viewing systems for landscape architecture, building construction and surface mining/reclamation. These might be characterized as "outdoor virtual reality" applications, and they are being built in 1991 at several sites. They differ from realtime simulation military simulation in several important ways:

- realism requirements.

  - Combat simulation cares little about the visual fidelity of buildings, trees etc; a schematic representation is sufficient.

  - Commercial visualization needs a lighting model; often environment-mapping is used to emulate the luster and reflectivity of ray traced imagery. An extensive surface texture map library is required.

- interactivity requirements.

  - Military training which incorporates combat engineer functionality will require that the dynamic terrain activity occur simultaneously with combat simulation. There is no difference between the creative, editing phase (e. g. building the emplacements with a team of combat engineer vehicles) and the wargaming phase.

  - For commercial customers, dynamic terrain construction could be performed in modes as simple as wireframe, as long as a subsequent free-play flyover of a high quality version of the resulting terrain is possible. The essence is that the users need to be able to interactively sculpt the landscape; e. g. to locate and configure the lakes and sand traps of a golf course.

- the size of the terrain database.

  - Distributed combat simulation must often provide several thousand square kilometers of play area. This quantity of information strongly drives the design of the database management system for an image generator.

  - Architectural and landscaping applications seldom exceed 4 square km. It is feasible to retain this database in the RAM memory of the imaging system.

In general, the dynamic terrain requirements for commercial virtual reality are within the scope of today's graphical workstations' capabilities. Because of the smaller scope of the requirements, special purpose image generators probably are not needed. Indeed, some of workstation prototypes built at IST to explore ideas for military DT will probably be further developed for these commercial purposes.

7. Conclusions

This paper has addressed two of the problems raised by dynamic terrain: the polygon and texture requirements for visualization. The broad conclusions of our study are:

a. An acceptable display for engineer, infantry, armor and artillary purposes is achievable by the mid-90's on simulators having approximately twice the graphical performance of current SIMNET units; but their visual systems will be substantially different from those in SIMNET.

b. Aircraft simulators of SIMNET/AirNet quality, already regarded as unacceptably crude by much of the aviation community, will be unable to render dynamic terrain in an acceptable fashion. Air/Land Battle simulation with dynamic terrain visible from the air will require aircrafc simulators of a different order of fidelity than ground-based simulators, because of the extended visual range of aircraft. Such fidelity may not be required for some training applications.

c. The minimal DT display system for military training requires an image generator capable of displaying 2000 to 4000 polygons at 15 frames/second per channel, equipped with appropriate database access and control algorithms in the front-end geometry processor.

d. For commercial dynamic terrain applications, it is likely that equipment equivalent to today's mid-performance graphical workstations will suffice. Some of their realtime visualization needs will be met running different software or hardware than the dynamic terrain systems which were used to sculpt the landscapes.

REFERENCES

Bess, Rick D. "Tradeoffs in the Configuration of Computer Image Generation Systems". *Proceedings of the Twelfth Industry/Interservice Training Systems Conference,* Orlando, FL, December 1990.

Biberman, Lucien M. *Perception of Displayed Information,* Plenum Press, New York-London, 1973.

Furness, Thomas A. III. "Harnessing Virtual Space." *Proceedings of the SID (Society for Information Display) 1988 International Symposium.* Playa del Rey, CA. May 1988.

Johnson, Richard S. "The SIMNET Visual System." *Proceedings of the Ninth Interservice Training Equipment Conference,* Washington, D. C. Nov. 30-Dec. 2, 1987.

Kass, Michael and Miller, Gavin. "Rapid, Stable Fluid Dynamics for Computer Graphics". Computer Graphics, 24:4, August 1990.

Moshell, J. Michael and Buckley, Robert. "Photographic Study of Simulated Dynamic Terrain." VSL Document VSL91.38, Institute for Simulation and Training, December 1991.

Simulated Dynamic Terrain:
A Photographic Study

J. Michael Moshell
Curtis R. Lisle
Robert Buckley

Institute for Simulation and Training
University of Central Florida

December 1991
VSL Document 91.38

This report is a part of the IST Dynamic Terrain Project, sponsored by the Army's Project Manager for Training Devices (PM-TRADE). For a broader picture of issues and approaches concerning the visualization of Dynamic Terrain, please see [Moshell 91].

The attached photos represent the results of varying three parameters:

- Pixel resolution (two values:)
  Level a: approximately 4.2 arc mins/pixel
  Level b: appxoximately 2.3 arc mins/pixel

- Lighting Model used (two values:)

  Flat Shaded
  Multigen Lighting Model (modified Gouraud) shaded

- Polygon Expenditure for Dynamic Terrain (three levels):
  No Dynamic Terrain
  256 Polygons for DT
  1050 Polygons for DT

These photographs were shot using a tripod mounted 35mm single lens reflex camera using Kodak Ektar 125 ASA 100 film, from the screen of a Silicon Graphics (SGI) 4D/240 workstation running the MultiGen database development system. Since the brightness of the monitor is a function of both its age and recent manual adjustments, the most reliable approach to determining good exposures is to "bracket" the timing. The best negative is then selected for printing.

MultiGen allows the user to set the angular field of view, and to stretch the screen window to achieve any desired physical size within the limits of the 19" CRT monitor. In order to establish a certain simulated viewing block aspect ratio and pixel density, a black cardboard mask was cut and attached to the screen.

The SGI's screen displays approximately 93 pixels per inch. In order to produce the desired Level a and Level b pixel densities, the camera is moved closer or farther from the CRT screen until the photographic image is the same size for either case. The pixel densities and scene setups were provided by a different project, and were opportunistically used for Dynamic Terrain visualization.

The reader should be warned that such simple photographic methods will inevitably lead to some loss of visual quality. Indeed, the on-screen images appear somewhat sharper than the photographs. However, the visibility assertions and tests referred to in the experimental paper cited above were based on the CRT images, not on the photographs.

It is interesting to compare the angular resolution of these images to that of the SIMNET system. The following table compares "Mark 1" (the low resolution simulation), SIMNET and "Mark 2" (the high resolution simulation).

	Pixels	**FOV (Deg.)**	**Pixels/Deg.**	**ArcMin/Px.**
MARK 1				
Vertical	130	9	14.47	4.15
Horizontal	560	40	13.95	4.30
SIMNET				
Vertical	128	7.5	17.07	3.52
Horizontal	320	20	16.00	3.75
MARK 2				
Vertical	250	9	27.90	2.15
Horizontal	930	40	23.25	2.58

As can be seen, the MARK 1 system is about 25% coarser than SIMNET, and the MARK 2 is about 40% finer. MARK 2 approximates the resolution that will be used for wide fields of view in the CCTT system currently being bid by vendors to PM-TRADE.

All scenes used the same background, consisting of approximately 330 polygons. For the Low DT Density photos shown as Figure 1, the dynamic terrain features were built as follows:

Feature	**Count**	**Polys/feature**	**Total Polygons**
Berm	6	12	72
Crater	6	12	72
Track	2	56	112
Total Dyn. Terrain			256

For the High DT Density photos shown as Figure 2, the dynamic terrain features were built as follows:

Figure 1:
MARK 1 (130 x 560 pixels)

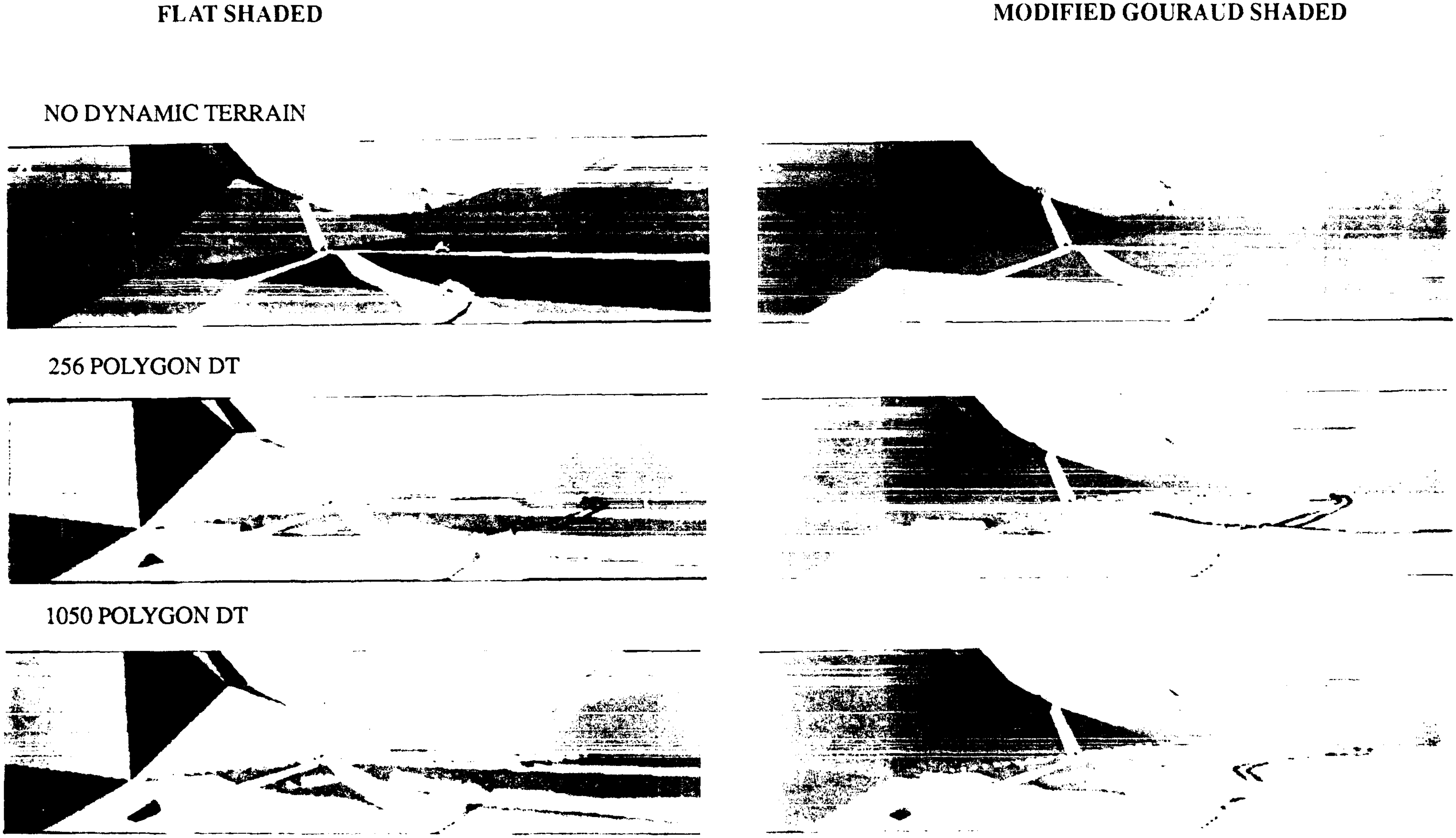
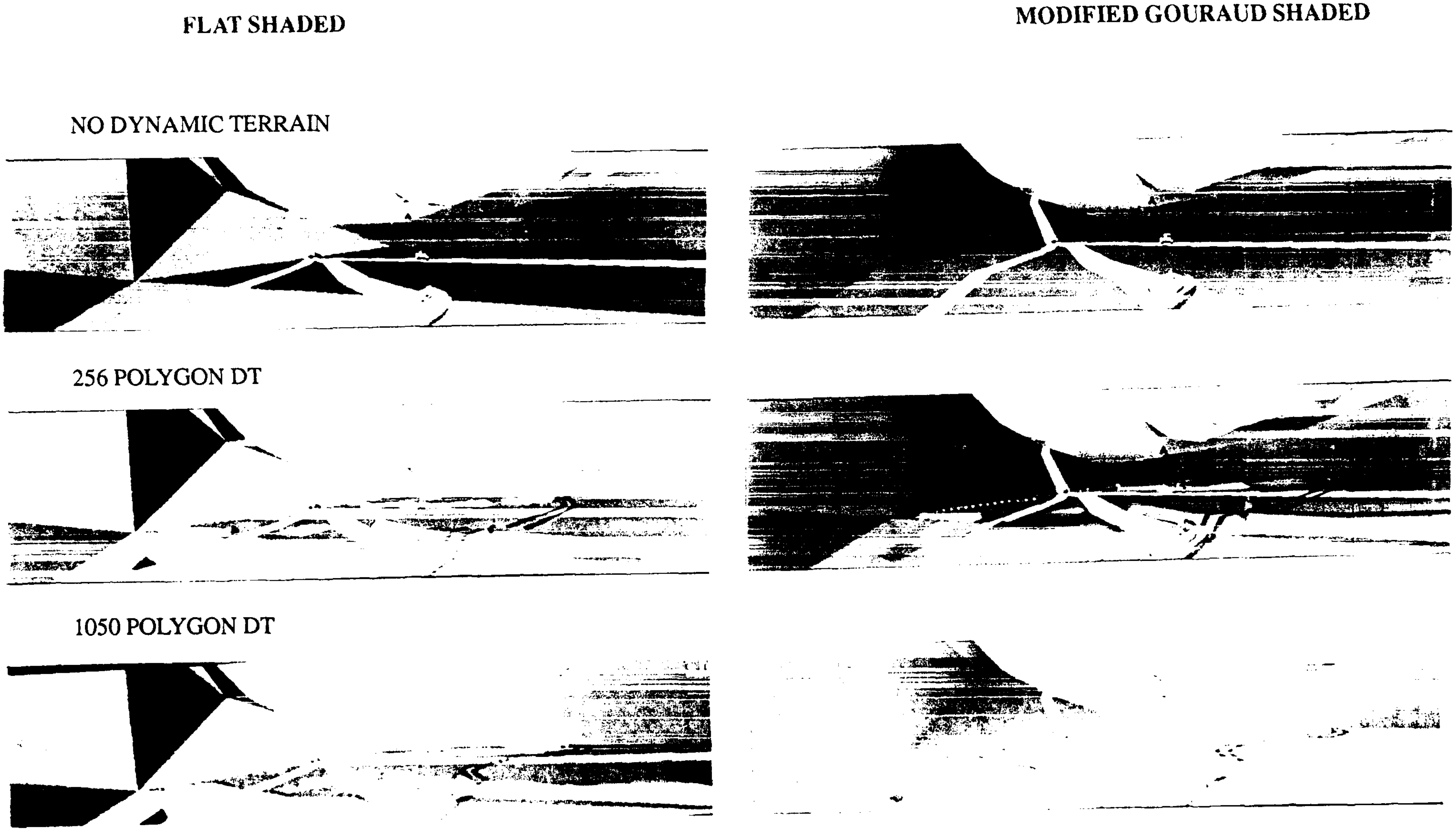


Figure 2:
MARK 2 (250 x 930 pixels)



FLAT SHADED

MODIFIED GOURAUD SHADED

NO DYNAMIC TERRAIN

256 POLYGON DT

1050 POLYGON DT

Figure [illegible]

256 [illegible]

Inward

Turret

Figure 4

1036 Polygon

Dynamic

Terrain

Feature	Count	Polys/feature	Total Polygons
Berm	10	56	560
Crater	6	48	288
Track	2	101	202
Total Dyn. Terrain			1050

From this series of photos, we have selected two for enlargement to illustrate the relative merits of 256 and 1050 polygons. These are the flat shaded MARK 2 photos, which are good approximations to what Dynamic Terrain in the BDS-D/ATTD system and the CCTT system can look like.

These appear as Figure 3 and Figure 4.

Flat shading is selected because its use is common in visual simulation, based on the widespread experience that the vertices enhance users' ability to orient themselves compared to Gouraud and other smooth shadings. Apparently the visible polygon edges in flat shaded images provide the human visual system with geometric cues which partially compensate for the artificial nature of the scene.

These images are not textured because our Laboratory workstation (Silicon Graphics 4D/240 GTX) does not presently support texture.

References

Moshell, J. Michael and Lisle, Curtis R. "The Visualization of Dynamic Terrain". VSL Document 91.20, Institute for Simulation and Training. 10 December, 1991.

Appendix E:

The ANIM Animation System

Dynamic Terrain Project
1990-91

ANIM - A Simple Animation Controller
VSL Memo 90.10

August 1990

Curtis Lisle

Visual Systems Laboratory
Institute for Simulation and Training
University of Central Florida

Abstract: ANIM is a general-purpose rendering package. It is designed to isolate the user from needing to understand both the area of three dimensional graphics transformations and the IRIS Graphics Library in order to get a graphical view from the IRIS workstation.

Table of Contents

Overview of ANIM

The Silicon Graphics Iris family of computers has considerable power for graphics-based applications since each system contains a graphics accelerator. Since the inception of IST/VSL, animation has been an area of research interest at the lab, but no general-purpose software had been developed to support this research.

To conduct research in animation, including emergent behavior of animated objects, behavioral simulations must be written and the final displays must be visually rendered. Before any behavioral simulations were developed, the problem of creating a general-purpose rendering environment was addressed.

The ANIM system on the Iris at IST/VSL is a result of this effort. ANIM supports an animation language which allows multiple objects in an environment along with a camera to view the objects. The positions, velocities, and orientations of any objects (including the camera) can be affected through the animation language interface. The commands in the language are designed to provide the user with tools to use when designing behavioral simulations.

A "User's Manual" for ANIM is included in this document. As of this writing, the ANIM system has been exercised in several applications including a terrain flyover, a battlefield simulation, and physical modeling of objects. Several of these applications are described in this document to serve as examples for any future work.

In the Design Issues section, the data formats used by ANIM are presented as well as more detailed issues about the structure of animated objects.

II. A User's Manual for ANIM

ANIM is a general-purpose rendering package. It is designed to isolate the user from needing to understand both the area of three dimensional graphics transformations and the IRIS Graphics Library in order to get a graphical view from the IRIS workstation.

As mentioned in the overview, ANIM's capability frees the user to concentrate on mathematical or behavioral models of their problem. When a graphical output is desired, simple commands can be used to position and render objects according to output from a user's simulation program. This can be supported as either a post-process rendering of an output file, or as a real-time rendering of commands from a user's program. Examples of these alternatives will be given later.

ANIM provides an environment where a number of polygonal objects (referred to as *AnimObjects*) can be initialized and then animated within a three-dimensional, Cartesian space. The shape, size, and behavior given to the objects is left to each user to define for his or her particular application.

A special need was identified during the development of ANIM to represent photorealistic terrain in addition to polygonal objects. To support the need, an object type called a *MeshObject* was created. These objects are kept in a separate list by ANIM, so multiple Meshes can be read in at any time.

The following paragraphs describe how to use ANIM. This package runs on the Silicon Graphics Iris 4D series of workstations.

Anim provides a set of commands which can be entered interactively, read from script files, or generated by external programs and piped into ANIM. The commands affect how all objects are displayed, or affect the state of a particular object being manipulated by the command. Example terminal sessions of ANIM are shown in the following sections to illustrate each type of interaction supported by ANIM.

ANIM provides a command line interface for the user. Commands from the command list can be typed one at a time as shown below:

```
% anim
anim:  readp diam.dat
anim:  circle right 10
anim:  draw
```

The available commands are covered in a later section of this document.

One of the commands provided by ANIM is .he *script* command which provides enough power to merit its description here. *Script* allows a series of commands (referred to as a script) to be read from an external file. This method is often used for standard, repeatable demos. The filename used for the script is completely up to the user. However, the naming convention where all script files have the extension ".ani" is encouraged. For example, "pooldemo.ani" and "RunADemo.ani" are filenames which conform to the standard. Below is an example of how the *script* command can be used:

```
% anim
anim:  script RunADemo.ani
anim:  exit
```

When using the script command, keep in mind that ANIM actually executes every command as if you typed them in at that point in time. This is an area where caution is suggested. If the *exit* command is included in the script, than ANIM will terminate, losing any objects which were read in and positioned. If the *readp* command, which reads an object into the ANIM data structure, is included in a script, then each time the script is run, ANIM will read a datafile – this will cause multiple copies of the same object with the same name in ANIM!

The multiple object problem can be easily solved by using separate initialization and animation scripts. The initialization is performed once and the animation can be replayed as many times as desired:

```
% anim
anim: script poolsetup.ani
anim: script runpool.ani
anim: script runpool.ani
anim: script runpool.ani
```

Since ANIM has a command-line interface, it can be executed directly from a Unix shell by redirecting input from an ANIM script file. Execution will return immediately to the shell when the script is completed because the end of the script file will signal ANIM with an EOF (end of file code) from Unix. This is useful for canned demos and is shown below:

```
% anim < file_of_commands.ani
%
```

Below is a list of the commands supported by ANIM with a short description of what each performs and how it is used.

animcam: Enable animation of the camera (viewpoint). When this option is enabled, the camera can be positioned like it was an object named "cam". Below is a sequence of commands which places the camera over a scene, and starts it moving:

```
readp terrain.dat
animcam  on
pos cam 100 100 50
rot cam -60 0 0
vel cam -0.5 -0.5 -0.5
play 100
animcam  off
```

axis: Display a coordinate axis at the global origin. This is helpful for orientation of the camera and objects:

```
ex:    axis on
       draw
```

circle: Move the viewpoint up,down,left,right around the origin. The value is expressed in degreees of rotation around the origin. This allows camera motion when the *animcam* mode is not active. The argument given is expected in degrees of rotation.

```
ex:    circle up 20        (orbit up around object 20 degrees)
       circle down 150
       circle left 15.2
```

delay: Set the redrawing speed of animation. A delay value specifies a waiting period between frames. If delay is never specified, it defaults to 1. This is useful if animation occurs too quickly. The values are relative and are not calibrated to an absolute clock. For example, a delay of 200 is slower than 100, but this is not an exact 200 milliseconds/frame. It is just slower.

```
ex.    delay 200
       delay 4000
       delay 1
```

display: Toggle whether a particular object is visible. Objects are identified by name. This is useful when a number of instances of a base object are made. The individual instances can be turned on or off. This was developed for a battlefield simulation where tanks disappear from the display when they are destroyed.

ex.

```
display diam off
display diam on
```

draw: Draw all the objects in the database without advancing time (moving any objects).

ex.

```
draw
```

echo: Display a message on the interactive window. This is usually used inside command language scripts. Only one word is allowed as an argument, so multi-word messages are connected by underlines by convention.

ex.

```
echo ***next_simulation_step***
```

exit: Exit ANIM and return to UNIX.

go: Move closer or farther from the origin. Values are expressed in "anim units" (floating values which do not directly correspond to inches, meters, etc.). This, like the *circle* command, is used when ANIM is not in the *animcam* mode.

ex.

```
go in 200
go out 100.4
```

instance: Make another object and have it use the polygon list from an existing object. This is useful for multiple objects with the same shape, for example, 15 pool balls on a table. The command is executed by giving the name of the existing object and the new object to create. New instances are created at the origin. After instancing, the new object can receive its own commands.

ex.

```
instance ball ball2
instance ball ball3
pos ball2 10.5 -0.5 3
```

play: Play frames of animation. All displayable objects (see *display* command) are drawn and the object's positions are updated whether they are displayable or not. The process is repeated until all requested frames have been displayed. ANIM keeps track of frame numbers since the beginning of each session. Therefore, a *play 100* command plays 100 additional frames and adds 100 to the ongoing frame count since the ANIM session was begun.

ex.

```
play 5
play 300
```

pos: Set the position of an object in absolute x,y,z coordinates. Each coordinate value is given in "anim units" and objects are referred to by name. This is an absolute position, so successive invocations replace each other (they are not additive). The position can be supplied using either integer or floating point units.

```
ex.   pos diam 10.5 10.3 -4.6
```

prompt: Turn the "anim: " prompt at the user's window on or off as desired. Disabling the prompt increases the animation speed substantially. The state of the prompt is maintained until a new command is issued. The default is "prompt on". If the prompt is off, it may be hard to determine when a script or sequence has completed since ANIM does not issue a new prompt at the user window.

```
ex.   prompt off
      prompt on
```

readh: Create a hierarchical object in ANIM. Read a a polygonal object from a datafile and create a new object which is below an existing object in the hierarchy. See the section on hierarchical objects for more detail about this process. In the example shown, a hierarchical arm is created by using the *readh* command. The syntax for the readh command is "readh <datafile> <object to attach to>".

```
ex.   readp shoulder.dat         (normal read of top object)
      readh arm1dat shoulder     (attach arm1 below shoulder)
      readh arm2.dat arm1        (attach arm2 below arm1)
      pos arm1 5 0 0             ( pos with respect to shoulder)
      pos arm2 5 0 0             (pos with respect to arm1)
```

readm: Read a mesh (2D array of elevations) into ANIM as a mesh object. The command's arguments are the mesh dimensions, the black & white image file, and the elevation file. The image file is warped to match the elevation at each point. As of the date of this paper, ANIM did not support motion for mesh objects, they can only be read in and displayed. Changes are necessary to the mesh displaying subroutines to support motion.

```
ex.   readm  meshname XSize YSize ImFile ElevFile
      readm  m 100 100 mesh.i mesh.e
```

readp: Read in a polygonal object. The object file format is described in another section of this paper. The object is added to ANIM's object list and can then be manipulated by name. The datafile name is the only argument. As can be seen from the data format description, a file can contain single or multiple objects.

```
ex.   readp sph.dat
```

rot: Rotate an object with respect to its local origin. This is an absolute rotation, so successive invocations replace each other (they are not additive). The rotation for an object is specified by giving its rotation in the **x, y, and z axes in degrees (positive or negative).**

ex.

```
rot ball 10.5 0 4.3
```

script: Read commands from a script file. The file is read until the end of file is reached, control is then returned to the interactive window. The only parameter needed is the script filename. A convention exists that script files end with the (.ani) extension, but ANIM supports reading from any Unix filename. Pathnames can be used, but the Unix tilde (~) is not supported.

ex.

```
script poolballs.ani
```

spin: Apply a relative rotation (around its local origin) on an object for every frame of animation played. After each frame, the rotation of the object around its local origin is updated by adding the parameters on the spin command to become the new rotation. For example, the command below increases the objects rotation by 2 degrees around X and 20.2 degrees around Z every frametime.

ex.

```
spin ball  1  0  20.2
```

vel: Assign a velocity to an object by name. Once given a velocity, ANIM will update the position of an object every frametime. The object will maintain its velocity until it is reassigned (to zero).

ex.

```
vel diam 0 0 0.5   (increase z each time)
```

where: Return the position and orientation of a named object. At the time of this writing, the display viewpoint has two sets of transformations applied to it: labeled *cam* and *camera*. The *camera* transformation is in polar coordinates and the *cam* transformation is in spherical coordinates relative to the polar position. See the Camera Animation section for a complete description.

ex.

```
where ball
where cam (returns x.y.z of cam xform)
where camera (returns dis,azi,elev,tw)
```

To fully use the capabilities provided by ANIM, it is necessary to understand how ANIM represents objects internally and what its external file formats are. In the following sections, some of this detail is provided.

Since ANIM is designed as a flexible animation system, attempts have been made to allow it to operate with as few or as many objects as the particular application demands. To support this design goal, objects are joined together with a linked-list as shown in Figure II.1. This allows the number of objects to be restricted only by the virtual memory size on the host computer.

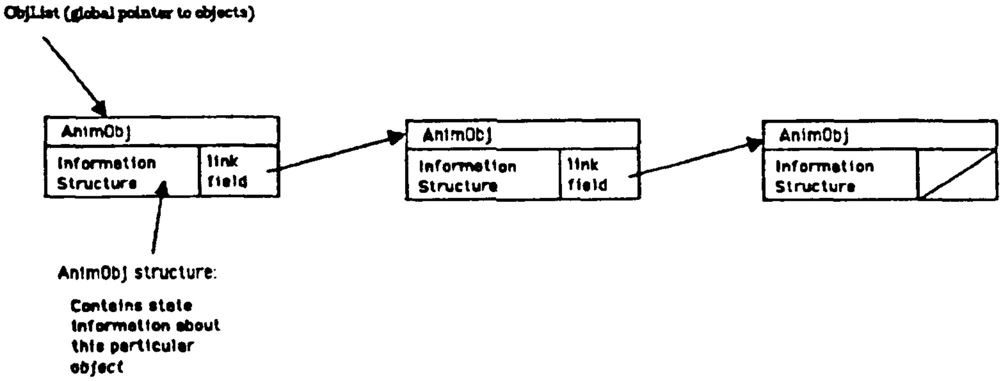


Figure II.1 - Anim objects in a linked list

Each object in ANIM (referred to as an AnimObject) is composed of a structure containing state information and lists for the object's polygons. The fields of the structure are shown here along with a description of each field:

```
struct AnimObj {
  int objnum;
  char name[15];
  struct AnimObj *SubObjList;
  int NumFaces;
  int display;
  struct face *faces;
  struct pos_info trans;
  struct rot_info rot;
  struct fly_info fly;
  struct AnimObj *link;
```

};

objnum - This field is not currently used by ANIM. Objects were initially going to be indexed. This could be used in the future to allow an index of pointers to speed access to AnimObjects.

name - A character string used to identify the AnimObject. All the commands use this field to retrieve the object pointer before affecting the object's state.

SubObjList - This is a list of other AnimObject types which are below the current object in the object hierarchy. See the Hierarchical object section for more details.

NumFaces - This field is not currently used by ANIM. It is always set to the number of polygons in the face list, but the rendering routines do not currently refer to this value.

display - A flag indicating whether this object is currently visible or invisible. When set to TRUE, the object is displayed whenever a *draw* or *play* is executed. This flag keeps its state until it is explicitly changed again.

faces - This is a pointer to the first polygon in a linked list of polygons. Each AnimObject can have an associated list of faces, which could be empty. The length of the polygon list is limited only by the memory capacity on the platform.

trans - structure which contains absolute position and velocity information for the AnimObject. This contains six subfields (x,y,z,dx,dy,dz).

rot - Similar to the trans structure, this structure contains fields specifying the absolute rotation and the rotational velocity of the AnimObject.

fly - This is not currently used by ANIM. This structure allows the Euler angles for yaw, pitch, and roll to be applied to an AnimObject. The ANIM system does not yet fully support this ability.

link - A link field to the next AnimObject maintained by ANIM in a global object list. The linking of objects is handled automatically by ANIM.

A diagram showing the data structure of a sample AnimObject is given in Figure II.2. This particular object has two polygons, each of which have three vertices. Each AnimObject will include a linked list of polygons (the *faces* list). In turn, each polygon record included in the *faces* list has a linked list of vertices hanging from it (the *verts* list comprising the vertices in that polygon). With this structure, an AnimObject can theoretically have an unlimited number of polygons and each polygon could have an unlimited number of vertices.

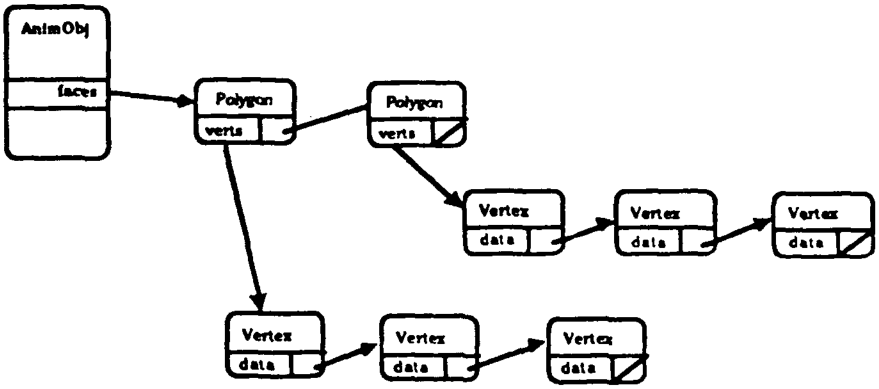


Figure II.2 - Polygon data in an AnimObject

Each MeshObject in ANIM is composed of a structure containing state information and pointers to image and elevation data for the MeshObject. The fields of the structure are shown here along with a description of each field:

```
/* single mesh object: */
struct MeshObj {
  char name[15];
  int numr, numc;
  unsigned char *im,*el;
  struct pos_info trans;
  struct rot_info rot;
  struct MeshObj *link;
};
```

name - a name field like that of an AnimObject. This could be used to find a pointer to a particular MeshObject, but is not currently supported by ANIM yet.

numr,numc - The number of rows and number of columns in the image and elevation files for the MeshObject.

im,el - Pointers to the image and elevation arrays. These are initialized when the MeshObject is read in by ANIM. See the description for the *readm* command.

trans,rot - Analagous structures to those of an AnimObject. These are not currently supported by ANIM. At the time of this writing, MeshObjects were stationary and are positioned parallel to the XY plane with their elevation values mapping to positions along the Z axis.

link - A pointer to the next MeshObject

ANIM uses several external file formats to provide its facility of loading objects and terrain. In the following sections, each of these file formats is presented. Any software projects developed can interface to ANIM by creating output data in the described formats.

ANIM objects are composed of any number of convex polygons with a separate color associated to each polygon. The objects can be read from one or more files using the *readp* and *readh* commands. The format of an ANIM object file is as follows:

```
integer: debug flag always=0
integer: number of ANIM objects in this file
(for each object in the file)
        character string: name of object
        integer: number of polygons in this object
        (for each polygon in this object)
                3 integers: RGB values for this polygon
                integer: number of vertices in this polygon
                (for each vertex)
                   float: Xvalue float: Yvalue float: Zvalue
```

A notated example file is shown below to further illustrate the format. The parsing is somewhat forgiving since blank lines can be embedded anywhere. Single tabs can be used as separators instead of spaces, however, at the time of this writing, ANIM could not support multiple tabs between fields. The explanatory comments are just to help illuminate the meaning of the data fields. Only the data (no comments) should be entered in actual datafiles.

```
0                    {no debug option}
1                    {one object in this file}
plane                {name to attach to the object}
3                    {the plane has three polygons}
200 50 50            {RGB values of the first (mostly red) poly}
3                    {3 vertices, first poly is a triangle}
5.0 0.0 0.0          {first vertex location}
0  -2  0             {second vertex location - float or integer ok}
0   2  0             {third vertex location}
200 200 200          {RGB of second poly}
```

```
3                       {second poly is a triangle}
0  0  0
0  0  1                 {vertex locations}
5  0  0
200 200 200             {RGB of third poly}
3                       {third poly is a triangle}
0  0  0
5  0  0                 {vertex locations}
0  0  1
```

A mesh is composed of two raw datafiles: the *image file* and the *elevation file.* Each one is composed of an array of bytes (characters in the C language) with a fixed-length header in the beginning of the file. A portion of the MeshObject read routine is shown to aid the creation of new MeshObject datafiles:

```
{
   ElevationFile = fopen(efname,"r");
   ImageFile = fopen(ifname,"r");

/* get memory space for the image files */
   MeshObj->el = (unsigned char *) malloc(numrows*numcols);
   MeshObj->im = (unsigned char *) malloc(numrows*numcols);

   /* read past the file header */
    fread(temp,1,12,ElevationFile);
    fread(temp,1,12,ImageFile);

   /* read file into MeshObject data structure */
    fread(MeshObj->el,1,numrows*numcols,ElevationFile);
    fread(MeshObj->im,1,numrows*numcols,ImageFile);
}
```

As mentioned briefly earlier in this document, ANIM supports hierarchical objects. The diagrams up to this point have shown that each AnimObject is composed of a list of polygons and a set of state variables. The examples given so far are actually all *flat objects* which is a subset of the possible types of objects supported by ANIM. In this context, *flat* means there are no hierarchical levels within the AnimObject.

To illustrate a *hierarchical object*, consider a car which wheels and a camera mounted to the roof on a swivel base. See Figure II.3. The car is composed of a body (which is one rigid piece), four wheels (which need to spin as the car rolls), and a camera (which can turn to the right or left around a center pivot). This can be represented as one object by making the wheels and the camera subobjects of the car itself.

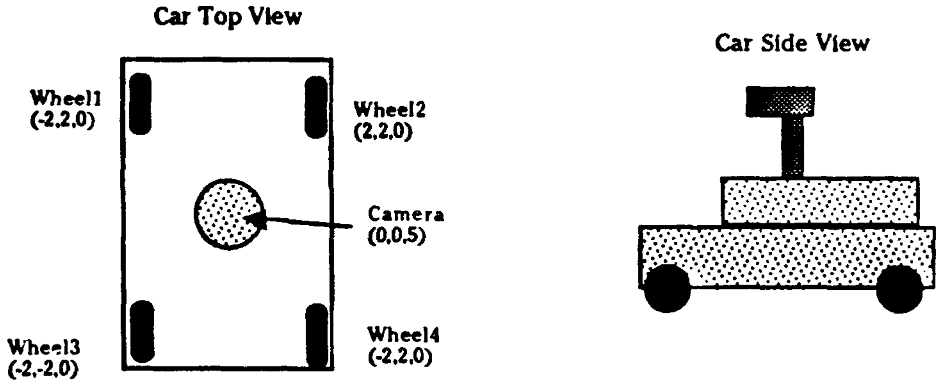


Figure II.3 - A Hierarchical Car

Whatever translation and rotation the car takes on will also happen to all its subobjects. Additionally, the translation and rotation given to subobjects acts with respect to the position of the car body – *consider state values of subobjects to be offsets from the state of their parent object.* For example, the needed translations for each wheel and the camera are given in parenthesis in Figure II.3 – they are specified as offsets from the center of the car (the parent object). Rotations given to the subobjects act around the their local centers, but their local coordinate axes are positioned with respect to the parent.

The car could be constructed several ways in ANIM. One way is shown by the set of following commands:

```
readp car.dat                  (create object “Car”)
readh wheel1.dat Car           (create object “Wheel1” under car)
readh wheel2.dat Car           (create object “Wheel2” under car)
readh wheel3.dat Car
readh camera.dat Car
pos Wheel1 -2.0  2.0  0.0      (position wheel1 with respect to car)
pos Wheel2  2.0  2.0  0.0
pos Wheel3 -2.0 -2.0  0.0
pos Wheel4  2.0 -2.0  0.0
pos Camera  0.0  0.0  5.0
```

The way shown above will result in the structure in Figure II.4 where the Car is the top level AnimObject and each subobject is placed in Car's SubObjList so they all take on the transforms placed on the Car in addition to their own.

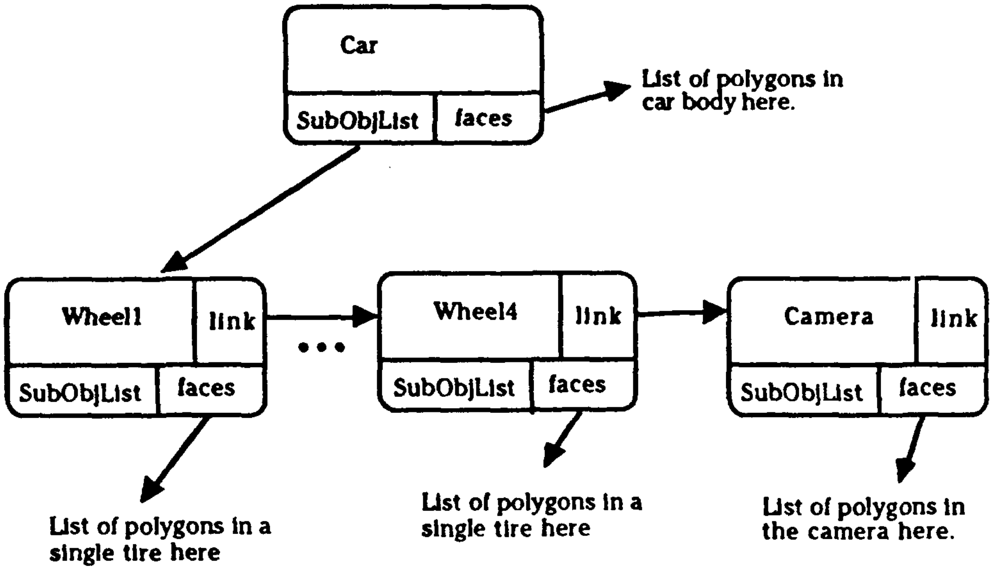

Car
SubObjList
faces
List of polygons in car body here.
Wheel1
link
SubObjList
faces
• • •
Wheel4
link
SubObjList
faces
Camera
link
SubObjList
faces
List of polygons in a single tire here
List of polygons in a single tire here
List of polygons in the camera here.

Figure II.4 - ANIM Structure for the Car

During the design of ANIM, the attempt was made to have all feautures implemented by using AnimObjects. This means, for examples, the camera is an AnimObject and the lights are AnimObjects.

Since lights are AnimObjects, the ANIM commands like *pos* and *vel* can be applied to a light. Lights can be placed anywhere in the scene and (if desired) given velocities to achieve desired lighting effects.

This area of ANIM is under development at the time of this writing. Therefore, no detail about this capability is included here. This section will be rewritten when the lighting model implementation has stabilized.

Since the camera is implemented as an AnimObject with an orientation and a position, then applying the reverse of the camera's translation and rotation to all polygonal objects gives the effect of camera motion. Rephrased, when the camera in ANIM is moved, what ANIM actually does is move every object in the scene by the opposite transform so that it *looks* like the camera moved.

This area of ANIM needs to be improved. Several features were planned here, but have never been implemented as of this writing. The remainder of this section describes the process which is necessary for the current version of ANIM to support camera animation.

The Dual Camera Problem: The camera exists as an AnimObject with the name *cam*. The camera object is added automatically by ANIM during system startup. Whatever cartesian transformations are applied to the *cam* object will have an affect on the view seen if the mode of ANIM is set to allow camera motion.

There is, however, a second camera object name called *camera* which has a position given in spherical coordinates (azimuth, incidence, distance from origin, twist). *Camera* is not actually an AnimObject, but the spherical transformations on *camera* are applied to all objects so the camera appears to be in this position.

A command entitled *animcam* was added to the commands and sets the status of a global flag. When set on, the translate and rotate commands applied to *cam* are performed before any objects are drawn. However, *the spherical transformations applied to "camera" are always in effect.* For this approach to work, the *camera* viewpoint must not be away from the origin or rotated – like what is applied by the *circle* and *go* commands when a transformation is applied to *cam*. If both cameras (*cam* and *camera*) are given transformations, the results can be unpredictable.

The example shown for the *animcam* command earlier involves the camera flying over a piece of terrain placed in the XY plane. For this example to work with ANIM in its present form, the *camera* transformation should be removed. The following commands provide a terrain flyover:

```
readp terrain.dat
where camera             (find and remove the spherical transform)
go in 40
circle left 100          (values dependent on the xform applied)
circle up 70
animcam  on
pos cam 100 100 50       (set cartesian position over terrain)
rot cam -45 0 0          (look down to see terrain)
vel cam -0.5 -0.5 -0.5   (set velocity of cam object)
play 100
animcam  off
```

Appendix F:

Constraints and Physical Modeling

Dynamic Terrain Project
1990-91

To: M. Moshell, C. Hughes

From: Julie Carrington

Date: February 4, 1991

Re: Physical modeling papers

Report number: 91.11, relating to project VSL91.2 - Constrained Dynamics

I have chosen eight papers to summarize in six summaries. In two cases papers were sufficiently similar to others by the same authors that I combined them in the same summary. See [3] and [5] in the summary list.

Several papers which I have not summarized are worth mentioning. I have listed four of these under "other relevant papers" in the summaries list. A complete survey on physical modeling in general probably would have to mention collision detection and response. Moore and Wilhelms' paper [7] appears to be something of a classic and is very commonly referenced. Another important topic in physical modeiing is that of the physical properties of materials. Terzopoulos and Fleischer [8] discuss viscoelasticity, plasticity, and fracture. Because [7] and [8] are somewhat off of the direct track of constrained dynamics, I have not included them in my summaries but both are clearly important papers.

Witkin, Fleischer, and Barr's paper [9] appeared at the same time as that by Isaac and Cohen [2]. Like Isaac and Cohen's paper, [9] has been widely referenced. I chose not to summarize [9] basically because I had already chosen three others by Witkin and two by Barr, all more recent than this one.

I had originally planned to include [10] which uses constraints to prevent movement of joints beyond their realistic limits. This is very similar to what is done in Isaac and Cohen's paper [2] which is much more readable. Zhao and Badler have what they call a real time system but by that they do not mean that they can, for example, make a figure walk in real time. It means only that they can manipulate a single joint over a small distance with a mouse.

With some exceptions, I have a clear enough understanding of the particulars of each system to implement simple examples. Before I can pursue this further, I need some deeper understanding of the principles of dynamics and mechanics which underlie these systems. There is a "big picture" here that I do not see. For example, both [5] and [4] contain a formulation of constraints which they call the Lagrange multiplier method. In both cases, a multiplier, λ is used to form a set of equations of motion with the proper number of equations and unknowns. Beyond that however, the two systems look and behave differently. In [5],(1) Witkin tells us that his system is related to [1]. I would like to better understand how all of these systems relate to each other; how they are different; where two systems can used together; and how they could be modified to suit particular applications.

In general I have used the same names for variables in my summaries as are used in the paper being summarized although I have tried to consistently use bold to denote vectors and capital letters for matrices. Upon printing, I discovered that while Greek letters show up bold on the screen, they do not print bold so I do have an accidental inconsistency where vectors are named with Greek letters.

Julie R. Carrington
February 4, 1991

SUMMARIES LIST

[1] Barzel, A.; Barr, A. "A Modeling System Based on Dynamic Constraints", A. Comp. Gr., Vol. 22, #4, 8/88, pp 179-188.

[2] Isaacs,P; Cohen,M, "Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions, and Inverse Dynamics", Comp.Gr., Vol.21, #4, 7/87

[3] (1) Pentland, Alex, "Computational Complexity versus Simulated Environments", Comp. Gr. Vol.24 #2, 3/90; and (2) Pentland.A; Williams, J. "Good Vibrations: Modal Dynamics for Graphics and Animation", Comp. Gr. Vol.23 #3, 7/89

[4] Platt,J; Barr,A.,"Constraint Methods for Flexible Models", Comp. Gr. Vol 22, #4, 8/88

[5] (1) Witkin,A.; Gleicher,M.; Welch,W.; "Interactive Dynamics", Comp.Gr. Vol.24, #2, 3/90, pp11-22 and (2) Witkin,A.; Welch,W.; "Fast Animation and Control of Nonrigid Structures", Comp.Gr. Vol.24, #4, 8/90, pp243-250.

[6] Witkin,A.;Kass, M., "Spacetime Constraints", Comp.Gr. Vol. 22, #4, 8/88.

OTHER RELEVANT PAPERS

[7] Moore,M;Wilhelms,J., "Collision Detection and Response for Computer Animation", Comp.Gr. Vol.22, #4, 8/88.

[8] Terzopoulos,D.; Fleischer,K., "Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture", Comp.Gr., Vol.22, #4, 8/88.

[9] Witkin,A.; Fleischer,K.; Barr,A., "Energy Constraints on Parameterized Models", Comp.Gr., Vol.21, #4, 7/87.

[10] Zhao,J.; Badler,N., "Real Time Inverse Kinematics with Joint Limits and Spatial Constraints", Tech Report MS-CIS-89-09, University of Pennsylvania.

Julie R. Carrington
February 4, 1991

[1] Barzel, A.; Barr, A. "A Modeling System Based on Dynamic Constraints", A. Comp. Gr., Vol. 22, #4, 8/88, pp 179-188.

Rigid body primitives such as rods, spheres, etc. move in a realistic way under gravity or other applied forces to meet constraints. Constraints can force bodies to attach themselves to each other or to points in space; to move along a user-defined path; or to align themselves by rotation.

For each body we have $\mathbf{F}_e$ and $\mathbf{T}_e$, vectors representing the external forces and torques. Using the deviation vector, $\mathbf{D}$ (i.e. the function which represents how far away from meeting the constraint the body is) we calculate $\mathbf{F}_c$. $\mathbf{F}_c$ is multiplied by a matrix G to produce the constraint force and by a matrix H to produce the constraint torque. G and H are 3 x f matrices where f is the number of degrees of freedom in the constraint force and they are intended to specify where on the body $\mathbf{F}_c$ is to be applied in order to produce the constraint force and torque. In the simple examples discussed in the paper, G is either the identity or the zero matrix and H is a matrix form ("dual", see appendix B) of the radial vector or the zero matrix. The net force and torque on body i are:

$$\mathbf{F}^i = \left(\sum G^i{}_j \, \mathbf{F}_{cj} \right) + \mathbf{F}^i{}_E$$

$$(1)$$

$$\mathbf{T}^i = \left(\sum H^i{}_j \, \mathbf{F}c_j \right) + \mathbf{T}^i{}_E$$ where the sums are over the constraints.

Because the deviation vector, $\mathbf{D}$, measures the distance from the constraint, where the constraint is met, $\mathbf{D} = 0$. $\mathbf{D}$ is a d dimensional vector with f degrees of freedom. For example, suppose we want to line up a unit vector $\mathbf{b}$ representing the radial vector fixed in a body with $\mathbf{N}$, a unit vector fixed in space. Then $\mathbf{D} = \mathbf{b} \cdot \mathbf{N} - 1$. Here f = 3 for the x, y, z components of $\mathbf{b}$ and d = 1 since the result is a scalar.

We describe the behavior of the constraint deviation with a linear combination of $\mathbf{D}$ and the first and second derivatives of $\mathbf{D}$:

$$\mathbf{D}^{(2)} + 2/\tau \, \mathbf{D}^{(1)} + 1/\tau^2 \, \mathbf{D} = 0, \; t \geq t^0 \qquad (2)$$

τ is a user defined time constant which determines how quickly the constraint will be met. The solution to this differential eouation is of the form $\mathbf{D} = c_1 e^{-t/\tau} + c_2 t e^{-t/\tau}$ and so approaches 0 at an exponential rate.

We differentiate the deviation vector twice in order to substitute into (2). When we find $\mathbf{D}^{(2)}$, the "acceleration" of the deviation, Barzel and Barr say that we should find a component (called $\Gamma\mathbf{F}$) to create the appropriate force, one (called $\Lambda\mathbf{T}$) to give us the proper torque and one independent of force and torque. This last (called β) consists of terms expressing rotational velocity and momentum and is due to the change from body to world coordinates. In other words, the forces and torques create rotation around the origin in body coordinates which is also the center of mass of the body. The movement of the body, however, also creates some rotation around the world coordinate system origin. This result is:

$$\sum_{\text{bodies } i} (\Gamma^i \mathbf{F}^i + \Lambda^i \mathbf{T}^i) + \beta + 2/\tau \, \mathbf{D}^{(1)} + 1/\tau^2 \, \mathbf{D} = 0 \qquad (3)$$

where $\mathbf{D}^{(2)} = \sum (\Gamma^i \mathbf{F}^i + \Lambda^i \mathbf{T}^i) + \beta$. Here Γ is generally the inverse mass matrix or 0, Λ is generally the radial vector times the inverse intertial tensor or 0, and β consists of leftover terms involving rotational velocity and momentum.

Next we substitute what we know about net force and torque from (1) into (3) to get:

$$\sum_{\text{bodies } i} \{ \Gamma^i [(\sum_{\text{constr } j} \mathbf{G}^i{}_j \, \mathbf{F}_{cj}) + \mathbf{F}^i{}_E] + \Lambda^i [(\sum_{\text{constr } j} \mathbf{H}^i{}_j \, \mathbf{F}_{cj}) + \mathbf{T}^i{}_E] \} + \beta +$$
$$2/\tau \, \mathbf{D}^{(1)} + 1/\tau^2 \, \mathbf{D} = 0$$

and rearrange for:

$$\sum_{\text{constr } j} [\sum_{\text{bodies } i} (\Gamma^i \mathbf{G}^i{}_j + \Lambda^i \mathbf{H}^i{}_j) \mathbf{F}_{cj}] + \sum_{\text{bodies } i} (\Gamma^i \mathbf{F}^i{}_E + \Lambda^i \mathbf{T}^i{}_E) + \beta +$$
$$2/\tau \, \mathbf{D}^{(1)} + 1/\tau^2 \, \mathbf{D} = 0 \qquad (4)$$

Notice that we have one of these equations for each constraint and each equation requires a sum over all constraints. This is because the force needed to enforce one constraint depends on the forces needed due to all of the other constraints.

Then our final system of equations is:

$$\boldsymbol{M}\ \boldsymbol{F}_{\mathrm{c}} + \boldsymbol{B} = 0 \qquad (4)$$

where $\boldsymbol{M}_{kj} = \sum_{\text{constr } j} (\Gamma^{i}{}_{k} G^{i}{}_{j} + \Lambda^{i}{}_{k} H^{i}{}_{j})$;

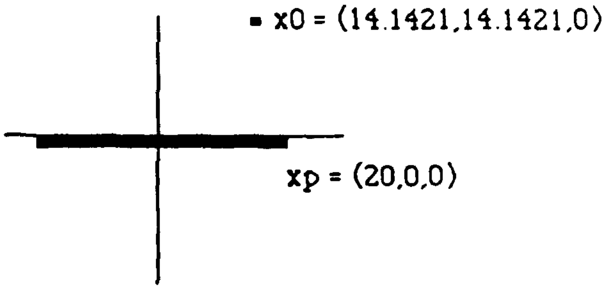
$\boldsymbol{F}_{cj} = \mathrm{F}_{cj}$;

and $\boldsymbol{B}_{k} = \sum_{\text{bodies } i} (\Gamma^{i} \mathrm{F}^{i}{}_{E} + \Lambda^{i} \mathrm{T}^{i}{}_{E}) + \beta + 2/\tau\ \mathrm{D}^{(1)} + 1/\tau^{2}\ \mathrm{D}$

Thus each element of $\boldsymbol{M}$ is a matrix with dimension d of constraint k by f of constraint j and is non-zero only if some body is affected by both constraints k and j. Each element of $\boldsymbol{F}_{\mathrm{c}}$ and $\boldsymbol{B}$ is a vector. Barzel and Barr suggest using singular-value decomposition (SVD) to solve the above equation because $\boldsymbol{M}$ can be singular (in an over or under constrained system) or non-square. Actually, $\boldsymbol{M}$ itself is square but its elements may be non-square matrices.

We look now at an example which illustrates how this system should work as well as questions which have arisen. The derivations associated with this example are long and look difficult but in part that is because I chose to follow every step and so make it easy to reproduce if not to follow initially.

We have a thin rod initially with its center of mass at the origin of the world coordinate system lying along the x-axis. We want to have point X_{p} attach itself to point X_{0}. Note that this is example 1 on page 183 of the paper.



The deviation vector, **D**, then is: $\mathbf{D} = \mathbf{X}_{\text{c?nter of mass}} + \mathbf{b} - \mathbf{X}_0$. For the following derivations of $\mathbf{D}^{(1)}$ and $\mathbf{D}^{(2)}$, we detail some notation and relationships most from Figure 16 on page 185 of the paper.

m = mass of the body. In our case m = 2.

R = Rotation matrix. Takes body coordinates to world coordinates. Initially the identity matrix.

I = Inertial tensor. In our case I_{body} is the diagonal matrix with [0.0667, 267.332, 267.332] as diagonal elements. I is the result of $RI_{body}R^T$.

X = Position of the center of mass. Initially (0, 0, 0)

b = Radial vector.in world coordinates Initially the same as $\mathbf{b}_{body}$ (20,0,0)

p = Momentum of the body. Initially (0, 0, 0)

L = Angular momentum of the body. Initially (0, 0, 0)

v = Velocity of the center of mass. Initially (0, 0, 0)

ω = Angular velocity. (0, 0, 0)

$\mathbf{v} = 1/m\mathbf{p} = d\mathbf{X}/dt$

$\omega = I^{-1}\mathbf{L}$

$\omega^*R = dR/dt$ (note: ω* is the dual of ω, see Appendix B page 186 of the paper)

$\mathbf{F} = d\mathbf{p}/dt$

$\mathbf{T} = d\mathbf{L}/dt$

We now do a detailed derivation of $\mathbf{D}^{(1)}$ and $\mathbf{D}^{(2)}$ following the derivations in Appendix B.2, page 186 of Barzel and Barr's paper. Recall that $\mathbf{D}^{(2)}$ should be of the form ($\Gamma\mathbf{F} + \Lambda\mathbf{T} + \beta$) as in (3).

$$\mathbf{D} = \mathbf{X} + \mathbf{b} - \mathbf{X}_0$$

$$\mathbf{D}^{(1)} = \frac{d\mathbf{X}}{dt} + \frac{d\mathbf{b}}{dt}$$

$$= \mathbf{v} + \frac{d(R\mathbf{b}_{body})}{dt}$$ from the above relations.

$$= \mathbf{v} + \omega^*R\mathbf{b}_{body}$$ above relations.

$$= \mathbf{v} + \omega \times \mathbf{b}$$ above relations and appendix B.1, page 186 of Barzel & Barr.

$$\mathbf{D}^{(2)} = \frac{d^2\mathbf{X}}{dt^2} + \frac{d^2\mathbf{b}}{dt^2}$$

$$= 1/m\mathbf{F} + \frac{d(\omega \times \mathbf{b})}{dt}$$ because $\mathbf{D}^{(1)} = \mathbf{v} + \omega \times \mathbf{b}$

$$= 1/m\mathbf{F} + \frac{d\omega}{dt} \times \mathbf{b} + \omega \times \frac{d\mathbf{b}}{dt}$$ using the chain rule.

$$= 1/m\mathbf{F} + \frac{d\omega}{dt} \times \mathbf{b} + \omega \times (\omega \times \mathbf{b})$$ from the derivation of $\mathbf{D}^{(1)}$.

$$= 1/m\mathbf{F} + \frac{d(\mathbf{I}^{-1}\mathbf{L})}{dt} \times \mathbf{b} + \omega \times (\omega \times \mathbf{b})$$ from the above list of relations.

$$= 1/m\mathbf{F} + (\frac{d\mathbf{I}^{-1}}{dt}\mathbf{L} + \mathbf{I}^{-1}\frac{d\mathbf{L}}{dt}) \times \mathbf{b} + \omega \times (\omega \times \mathbf{b})$$ chain rule.

$$= 1/m\mathbf{F} + (\frac{d\mathbf{I}^{-1}}{dt}\mathbf{L} + \mathbf{I}^{-1}\mathbf{T}) \times \mathbf{b} + \omega \times (\omega \times \mathbf{b})$$ above relations.

$$= 1/m\mathbf{F} + (\frac{d(\mathbf{R}\mathbf{I}^{-1}{}_{\mathbf{body}}\mathbf{R}^{\mathbf{T}})}{dt}\mathbf{L} + \mathbf{I}^{-1}\mathbf{T}) \times \mathbf{b} + \omega \times (\omega \times \mathbf{b})$$ from the derivation of $\mathbf{D}^{(1)}$

$$= 1/m\mathbf{F} + [(\frac{d\mathbf{R}}{dt}\mathbf{I}^{-1}{}_{\mathbf{body}}\mathbf{R}^{\mathbf{T}} + \mathbf{R}\mathbf{I}^{-1}{}_{\mathbf{body}}\frac{d\mathbf{R}^{\mathbf{T}}}{dt})\mathbf{L} + \mathbf{I}^{-1}\mathbf{T}] \times \mathbf{b} + \omega \times (\omega \times \mathbf{b})$$ chain rule

$$= 1/m\mathbf{F} + [(\omega^*\mathbf{R}\mathbf{I}^{-1}{}_{\mathbf{body}}\mathbf{R}^{\mathbf{T}} + \mathbf{R}\mathbf{I}^{-1}{}_{\mathbf{body}}\mathbf{R}^{\mathbf{T}}\omega^{*\mathbf{T}})\mathbf{L} + \mathbf{I}^{-1}\mathbf{T}] \times \mathbf{b} + \omega \times (\omega \times \mathbf{b})$$ above relations.

$$= 1/m\mathbf{F} + [(\omega^*\mathbf{I}^{-1} + \mathbf{I}^{-1}\omega^{*\mathbf{T}})\mathbf{L} + \mathbf{I}^{-1}\mathbf{T}] \times \mathbf{b} + \omega \times (\omega \times \mathbf{b})$$ definition of $\mathbf{I}^{-1}$

$$= 1/m\mathbf{F} + [\ \omega^*\omega + \mathbf{I}^{-1}\omega^{*\mathbf{T}}\mathbf{L} + \mathbf{I}^{-1}\mathbf{T}] \times \mathbf{b} + \omega \times (\omega \times \mathbf{b})$$ above relations.

$$= 1/m\mathbf{F} + [\ \mathbf{I}^{-1}(\omega^{*\mathbf{T}}\mathbf{L} + \mathbf{T})] \times \mathbf{b} + \omega \times (\omega \times \mathbf{b}),\ \omega^*\omega = 0,$$ appendix B.1 page 186, Barzel & Barr

$= 1/m\mathbf{F} + [\,\mathbf{I}^{-1}(\mathbf{L} \times \omega + \mathbf{T})] \times \mathbf{b} + \omega \times (\omega \times \mathbf{b})$ appendix B.1.

$= 1/m\mathbf{F} + (\mathrm{b}^{*T}\mathrm{I}^{-1})\mathbf{T} + \mathrm{b}^{*T}\mathrm{I}^{-1}(\mathbf{L} \times \omega) + \omega \times (\omega \times \mathbf{b})$ appendix B.1.

We now have $\mathbf{D}^{(2)}$ in the form $(\,\Gamma\mathbf{F} + \Lambda\mathbf{T}\,) + \beta$ where
$\Gamma = 1/\mathrm{m}$
$\Lambda = (\mathrm{b}^{*T}\mathrm{I}^{-1})$
$\beta = \mathrm{b}^{*T}\mathrm{I}^{-1}(\mathbf{L} \times \omega) + \omega \times (\omega \times \mathbf{b})$

Since $\mathbf{F} = \mathbf{F}_c + \mathbf{F}_E$, $G = 1$ and since $\mathbf{T} = \mathbf{b} \times \mathbf{F}_c + \mathrm{T}_E$, $H = b^*$. Notice that this derivation agrees precisely with the definitions of all of these variables given for this example in Example 1, page 183 of the paper.

We use $\tau = .2$ and get on the first iteration $\mathbf{F}_c$ = [-292.895, 177.107, 0]. This produces a torque in the right direction but there are several problems. For one thing, this $\mathbf{F}_c$ increases with every iteration which means that the rod spins and never comes to rest. In *addition*, $\mathbf{F}_c$ is to be multiplied by $\mathbf{b}$ for the torque but appplied to the body as force. Clearly, applying this force to the center of mass sends the rod off quickly to the left. Notice that if gravity is included in the model, the $\mathbf{F}_c$ is simply [-292.895, 186.907, 0], i.e., 9.8 more in the y direction and so exactly the same net force. In general these amounts for $\mathbf{F}_c$ seem excessive even on the first iteration and certainly they should not increase on successive iterations. Finally, I question their applying an arbitrary force initially in each of their examples. Why should this be necessary and what, exactly, constitutes an arbitrary force? Does gravity to the job? Several times, they refer to this arbitrary force as $\mathbf{F}_c$.

Note that everything works beautifully when we want to attach a sphere to a point in space, i.e., when there is no torque involved. It works well even though where the sphere has the same mass as the rod, the calculated $\mathbf{F}_c$ is nearly the same. This makes me wonder if in the rod the magnitude of the constraint force is correct but instead of applying it both as torque at the end of the rod and as force at the center of mass, it should be distributed through the rod. Possible explanations for the fact that $\mathbf{F}_c$ increases rather than decreases in the rod are a sign error or a coordinate system error.

This paper is hard to understand without a mechanics background but is worth the time and effort. The authors are well known, this paper is frequently cited, and even the example (the chain attaching itself to the trap door) is famous. The advantage of this system over Witkin's [5] is that the constraints need not be initially fulfilled. The body moves to meet them allowing, for example, a space station to assemble itself. One disadvantage is that it is not interactive although constraints can be made active at a specified time during the simulation. A more severe limitation is that it deals only with rigid bodies and any kind of realistic simulation must include realistic (therefore flexible) materials. Witkin's system does allow flexible bodies although his example is an extremely simple two dimensional model probably because of the computational complexity of flexible body dynamics [5].

Julie R. Carrington
February 4, 1991

[2] Isaacs,P; Cohen,M, "Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions, and Inverse Dynamics", Comp.Gr., Vol.21, #4, 7/87

The DYNAmic MOtion (or DYNAMO) system described by Isaacs and Cohen appears to be one of the original inspirations for many of the physical modelling systems described in these summaries. The DYNAMO system tries to incorporate the realism that stems from using physics with the level of control from traditional animation techniques.

DYNAMO performs dynamic simulation on linked figures. A link is a rigid body with size shape and mass. Links are connected to each other by joints in a tree structure. For example, the human body can be considered a linkage where the hips are the root link to which are connected three main branches: the upper body, the right leg and the left leg. Each link has between one and six translational and rotational degrees of freedom and the joint may have limits which prevent movement of some degree of freedom beyond a certain point. The lower arm, for example, is considered to have a single degree of freedom relative to the elbow but that joint prevents movement beyond a certain arc. Joints also have associated springs and dampers and the linkage responds to external forces and torques.

The three control mechanisms contained in the DYNAMO system are: 1. "kinematic constraints" allow some motion to be explicitly specified while other portions of the linkage respond in a natural way to the forces induced by the specified motion; 2. "behavior functions" to force the body to react to its surroundings; 3. "inverse dynamics" to calculate the forces which induce the specified motion.

The equations of motion are expressed either as forward dynamics:

$$\mathbf{q}'' = A^{-1}\mathbf{B} \qquad (1)$$

or as inverse dynamics:

$$\mathbf{B} = A\mathbf{q}'' \qquad (2)$$

where **B** is the force vector, [A] is the generalized mass matrix, and q is the acceleration vector. Internal forces are spring and damper forces of the joints:

$$\mathbf{F}_{spring} = k_{spring} \ * \text{ offset from center position}$$
$$\mathbf{F}_{damper} = k_{damper} \ * \text{ velocity of degree of freedom.}$$

Because we allow both forward and inverse dynamics, we have a situation where in some cases the acceleration is given and the force is unknown and in others, the acceleration is unknown and the force is given as follows:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \begin{bmatrix} q_1'' \\ X \\ q_3'' \\ X \end{bmatrix} = \begin{bmatrix} X \\ B_2 \\ X \\ B_4 \end{bmatrix} \qquad \text{(X's are unknowns)}$$

In this case, we first solve for the unknown q_i"'s by moving all terms involving known q_i"'s to the right side of the equation, removing those rows and columns from [A], and solving the reduced system. This procedure is outlined in an understandable way on page 219 of the paper. Once all of the accelerations are known, we can solve for the unknown forces by substituting into the original equation.

The derivation of the equations of motion in contained in the appendix. We will follow along here. The general equation is based on D'Alembert's principle of virtual work which says basically that the sum of the work of applied and internal forces and torques will be equal and opposite to the work of changes in momentum. Thus, for a system of n links:

$$\sum_{i=0}^{n} [\mathbf{dr}_i(\mathbf{F}_i - m_i\mathbf{r}_i'') + \mathbf{d}\theta_i(\mathbf{M}_i - \mathbf{L}_i') + \mathbf{dW}_i] = 0 \qquad (3)$$

Suppose link i is an entirely unconstrained particle. Then there are no internal forces; the applied force, $\mathbf{F}_i$, equals the mass times the linear acceleration; and $(\mathbf{F}_i - m_i\mathbf{r}_i'') = 0$. Similarly, the applied torque, $\mathbf{M}_i$ equals the change in angular momentum, $\mathbf{L}_i'$; and $(\mathbf{M}_i - \mathbf{L}_i') = 0$. Of course, $\mathbf{dW}_i$ (work done by internal forces) will also be 0. Note here that $\mathbf{dr}_i$ refers to the change in linear displacement and $\mathbf{d}\theta_i$ refers to the change in angular displacement.

Notice that in these n equations we have 6n unknowns: r_x'', r_y'', and $\mathbf{r}_z''$ for the linear acceleration and θ_x'', θ_y'', and θ_z'' for the angular acceleration in each equation. In addition, we need constraint equations specifying disallowed directions. Recall that each joint has limited degrees of freedom. We can reduce the number of unknowns and eliminate these constraint equations if we express the equations of motion in terms of the degrees of freedom of the links. To do this, we use four generalized coordinate vectors, each of which has one entry for each degree of freedom.

q	: A vector of positions (either linear or angular)
q'	: A vector of velocities.
q''	: A vector of accelerations (unknowns).
dq	: A vector of virtual displacements.

By "virtual displacement" I understand that we assume that a particle moves slightly in a direction which is disallowed by the constraints in order to calculate an internal force to counter that supposed movement.

Now we derive matrix equations for each of the vector quantities, $\mathbf{d\theta}$, $\mathbf{dr}$, θ'', $\mathbf{r}''$ and $\mathbf{dW}$, in terms of our generalized coordinates in order to substitute back into equations (3).

$$\mathbf{d\theta} = -T^T p^T \mathbf{dq} \qquad (4)$$

T is a connectivity matrix.

p is a matrix of unit axes of rotation of rotational degrees of freedom (or 0 for linear degrees of freedom).

$$\mathbf{dr} = [\, p \times T\,(C + Z)\,T - kT]^T \,\mathbf{dq} \qquad (5)$$

C is a matrix of distances from joints to the centers of mass of the links which they connect.

Z is a matrix of current displacement vectors along translational degrees of freedom (or 0 for rotational degrees of freedom).

$$\theta'' = -T^T (p^T \mathbf{q}'' + f) + \theta_0'' \qquad (6)$$

f is a matrix expressing the effect of the angular velocity of one degree of freedom on the angular acceleration of another.

θ_0'' is a vector of coordinates expressing the angular acceleration of the inertial frame of reference.

$$\mathbf{r}'' = [p \times T\,(C + Z)\,T - kT]^T\,\mathbf{q}'' + U \qquad (7)$$

U is an array in which each entry corresponds to the relationship of one link to known quantities in the system.

$$\mathbf{d\,W} = -dq^T\,(k \cdot \mathbf{X} + p \cdot \mathbf{Y}) \qquad (8)$$

X is a vector of internal applied forces.
Y is a vector of internal applied torques.

For more details, we are referred to pages 147-163 of this paper's reference [15]. Although that book is not in our library, we surely would need more details in order to fully understand the above list of formulas. The definition for U, for example, tells us nothing about its derivation or its use.

An interesting thing that falls out of this derivation is that if we substitute equations (4) through (8) into equation (3) then isolate the terms involving **q**", we get:

$$\mathbf{d\,q}^T\,(-A\ \mathbf{q}'' + \mathbf{B}\ ) = 0 \qquad (9)$$

where $A = [\ p \times T\,(C + Z)\,T - kT] \cdot m[\ p \times T\,(C + Z)\,T - kT]^T +$
$(pT) \cdot \mathbf{J} \cdot (pT)^T$. J is the moment of inertia tensor.

and
$$\begin{aligned} B = [\ p \times T\,(C + Z)\,T - kT] \cdot (F - mU) \\ - pT\,.\,[\mathbf{M} + \mathbf{J} \cdot (T^T f - \theta_0'') - \theta' \times J \cdot \theta'\ ] \qquad (10) \\ - k \cdot \mathbf{X} - p \cdot \mathbf{Y}. \end{aligned}$$

Isaacs and Cohen tell us that because the motion of the generalized coordinates is independent, we can drop the term $\mathbf{d\,q}^T$ out of equation (9). (I would have thought that where each acted independently, the virtual displacement would be 0 and so **d q** would be 0. If **dq** can be dropped, it must be non-zero. Clearly, I need a better understanding of virtual work.) We are left with force equals mass times acceleration or -Aq" + **B** = 0. This is the only paper which gives an explicit formula for a generalized mass matrix although I am still not sure what each of the terms is.

It is worth noting that this is not (and is not intended to be) a real time animation system. We are told that the computation time is inversely proportional to the time step and exponentially dependent on the number of degrees of freedom. Thus, for example, a human figure with 39 degrees of freedom executing a kick took 30 minutes of CPU time to produce one second of simulation time.

Julie R. Carrington
February 4, 1991

[3]

1. Pentland, Alex, "Computational Complexity versus Simulated Environments", Comp. Gr. Vol.24 #2, 3/90
2. Pentland, Alex, "Good Vibrations: Modal Dynamics for Graphics and Animation", Comp. Gr. Vol.23 #3, 7/89

The ideas that Pentland presents here are exciting. His system. ThingWorld, allows use of flexible models and constraints which may or may not be initially met. Despite all of the complexity inherent in modeling flexible materials, he achieves his simulations in real time.

Pentland tells us that rigid body dynamics can never realistically allow us to model interaction such as friction or collision between bodies. The usual alternative is the finite element method. The problem with finite element is its cost. Pentland says that the computation time is $O(n^3)$ where n is the number of nodes in the object. Notice that the simple bar shown on page 190 of (1) has approximately 100 nodes. An additional problem with finite element is that many small, frequent displacements are produced. These must be tracked using a small time step and averaged to produce an image more or less free of aliasing artifacts.

Physically based constraint systems such as that suggested by Barzel and Barr [1] are another problem. Pentland tells us that the complexity of these systems is $O(lk^2)$ where k is the number of constrained parameters and l is the number of constraints. Again. this is prohibitively expensive for a real time system which purports to model more than a simple toy world.

The solution to these problems is the technique of modal analysis instead of finite element analysis. The basic idea is that the deformation of an object under force can be described in terms of different orders of modes. Figure 1 on page 216 of (2) shows in the first picture a cylinder. The next two pictures show first order or linear deformations in which there is no bending or twisting. The first picture on the bottom row illustrates a second order deformation (bending) and the last two show combinations of first and second order modes.

Pentland's description of the finite element method (page 216 of (2)) is concise and easy to understand. An object is considered to be made of a three dimensional grid of nodes which under forces are displaced from the center of mass of the object. The basic equation is:

$$Mu'' + Du' + Ku = \mathbf{f}$$

where **u** is a vector of size 3n describing the displacements of the n nodal points in the x, y, and z directions. M is a mass matrix, D is a damping matrix, and K describes the material stiffness between points. The 3n element vector **f** describes the forces acting on the nodes.

Modal analysis is similar except that by diagonalizing the three matrices, M, D, and K, we are able to transform the above equation into 3n independent differential equations each of which describes the time course of one "vibration mode". I understand a single vibration mode to be the way a single node moves in a single direction (i.e., x, y, or z) away from its equilibrium point. In most of Pentland's explanations of how the modes are used, this interpretation makes sense. I am not clear, however, on the connection between this concept and the idea of different orders of modes as expressed by Figure 1.

Although the modal method is simpler and faster than finite element, its real advantage lies in the fact that because we are working with small independent equations, we have more control over the computation.

High frequency vibration modes have little impact on the shape of the object and so can be discarded, greatly reducing the amount of computation which must be done. If my understanding of vibration modes is correct, it is easy to see that the high-frequency modes would be those with low amplitude and thus little impact.

Not only does eliminating the high frequency modes reduce the number of equations to be solved but it allows us to use larger time steps. This is because the time step must be inversely proportional to the highest frequency. It makes sense that the time step must be small enough to capture the movement of the highest frequency mode.

In finite element the matrices M, D, and K must be periodically recomputed. How often depends on the highest frequency modes.

Thus, more savings are realized in the modal method by less frequent recalculation of these matrices.

As mentioned above, temporal aliasing is also a result of the high frequency modes in finite element. Because we can ignore these modes, a side effect of using modal analysis is that this problem is eliminated along with the time that would otherwise be needed to track and average these small displacements.

Assuming that our goal in modelling a flexible object is to give it a realistic appearance, we can even avoid computing the simplified equations obtained by diagonalizing the matrices M, D, and K. This is because we can precompute deformation modes of some regular shape such as a rectangular solid with approximately the same moments of inertia as the object. As illustrated by Figure 3 in (2), page 218 there is virtually no difference in the appearance of the deformations calculated by the actual deformation modes of the object and those precalculated using a rectangular solid.

To explain why I chose this paper to summarize, I would like to quote Alan Barr from SIGraph '89, the Panel Proceedings called "Physically-Based Modeling: Past, Present, and Future". In reference to Pentland's work on modal analysis, Barr said, "Now that's good stuff." (pg. 209)

Julie R. Carrington
February 4, 1991

[4] Platt,J; Barr,A.,"Constraint Methods for Flexible Models", Comp. Gr. Vol 22, #4, 8/88

Platt and Barr describe models made of flexible materials such as putty or clay and some of the kinds of constraint systems appropriate for such materials. Flexible models should both move and deform in a realistic and interesting way. For example, a collision should cause a body to squash while retaining its volume and a large deformation should cause the material to not return to its rest shape.

Flexible bodies are discretized into mass points via the finite element method. (An easy to understand explanation of the basic finite element method is contained in [3].) Because many mass points imply many state variables, constraint methods such as the "dynamic constraints" outlined in [1] are hard to use. This paper discusses simpler methods appropriate for flexible models.

The three constraint methods used by Platt and Barr are the penalty method, reaction constraints and augmented Lagrangian constraints. The penalty method and augmented Lagrangian constraints are both examples of optimization techniques. A very clear description of optimization in general and the penalty method specifically is contained in J. Burg's "Constraint-Based Programming: A Survey", pages 83 - 93.

A reaction constraint operates on a single mass point and calculates a final output force based on the net force resulting from other constraint methods and external forces. Therefore the reaction constraint is the last one calculated in a time interval so only one can be applied at a time to any given mass point. The reaction constraint calculates an unconstrained force by projecting out all components of the net force which would tend to violate the constraint. Next a constrained force is calculated and added to the unconstrained force to get the desired output force.

The constrained force, $\mathbf{F}_c$, is calculated from **D**, the deviation vector, i.e., the vector which points from the mass to where it should be. Then to set **D** to zero, we have:

$$\mathbf{F}_c = \mathbf{kD} + c\frac{d\mathbf{D}}{dt} \tag{1}$$

where k expresses how strongly we want to enforce the constraint and c is the damping constant. Where $c = \sqrt{2}k$, the constraint is fulfilled with critically damped motion.

We will follow the example in appendix B which forces a mass point to lie on a plane. The normalized plane equation is:

$$P(\mathbf{x}(t)) = Ax(t) + By(t) + Cz(t) + D = 0.$$

Thus the normal vector, $\mathbf{n} = (A\ B\ C)^T$ and since we want the distance from the position of the mass point, $\mathbf{X}$, to the plane to be 0, we have a deviation vector:

$$\mathbf{D} = -\,\mathbf{n}\ P(\mathbf{X}) = -\,\mathbf{n}\ (\ AX_x + BX_y + CX_z + D)$$

and where v is the velocity of the mass point:

$$\frac{d\mathbf{D}}{dt} = -\,\mathbf{n}\,\frac{dP}{dt} = -\,\mathbf{n}\ (\ Av_x + Bv_y + Cv_z\ ).$$

To produce the unconstrained force, we want to cancel any portion of the input force which is acting in the same direction as the normal to the plane.

$$\mathbf{F}_{unconstrained} = \mathbf{F}_{input} - (\ \mathbf{F}_{input} \cdot \mathbf{n}\ )\,\mathbf{n}.$$

$$\mathbf{F}_{constrained} = (\ \mathbf{kD} + c\ \frac{d\mathbf{D}}{dt}\ ) = -\ (k\ P(\mathbf{X}) + c\,\frac{dP}{dt}\ )\,\mathbf{n}.$$

Finally, the sum of $\mathbf{F}_{unconstrained}$ and $\mathbf{F}_{constrained}$ gives us the output force to be applied to the mass point.

Next we look at the Lagrangian constraint method which, again, is a constrained optimization technique similar to the penalty method. Notice that as far as I can determine, this Lagrangian method has little in common with that proposed by Witkin in [5].

A constrained optimization procedure must locally minimize some function f(x) subject to constraint g(x) = 0. We want to find a function which we can minimize without constraints which will give us the needed point on f(x). A critical point of the energy function:

$$\mathcal{E}_{Lagrange}(\mathbf{X}) = f(\mathbf{X}) + \lambda g(\mathbf{X}) \qquad (2)$$

gives us a solution to $f(\mathbf{X})$ subject to $g(\mathbf{X}) = 0$. The extra variable, λ, gives us an extra degree of freedom as illustrated by Platt and Barr in a simple example. Suppose we need a point as close as possible to the origin with the constraint that the point lie on the line $x + y = 1$.

$$\mathcal{E}_{Lagrange}(\mathbf{X}) = x^2 + y^2 + \lambda(x + y - 1)$$

Taking the partial derivatives with respect to each variable, we get three equations in three unknowns.

$$\frac{\delta \mathcal{E}_{Lagrange}}{\delta x} = 2x + \lambda = 0$$

$$\frac{\delta \mathcal{E}_{Lagrange}}{\delta y} = 2y + \lambda = 0$$

$$-\frac{\delta \mathcal{E}_{Lagrange}}{\delta \lambda} = x + y - 1 = 0$$

Applying gradient descent to equation (2) produces the differential equations:

$$v_{x_i} = -\frac{\delta \mathcal{E}_{Lagrange}}{\delta x_i} = -\frac{\delta f}{\delta x_i} - \lambda \frac{\delta g}{\delta x_i} \qquad (3)$$

$$\lambda' = -\frac{\delta \mathcal{E}_{Lagrange}}{\delta \lambda} = -g(\mathbf{X}). \qquad (4)$$

However, we find that we must make a sign change in (4) in order to have a system which behaves in a stable manner. Thus,

$$\lambda' = \frac{\delta \mathcal{E}_{Lagrange}}{\delta \lambda} = g(\mathbf{X}) \qquad (5)$$

gives us our change in λ.

To continue with the preceding example, suppose we have a mass point initially at the position (1, 1) and again we want to move it to the point on the line x + y = 1 which is closest to the origin. Then in each iteration, i:

$$\lambda_i' = x_{i-1} + y_{i-1} - 1$$
$$\lambda_i = \lambda_{i-1} + \lambda_i' \, dt$$
$$v_x = -2\, x_{i-1} - \lambda_i$$
$$v_y = -2\, y_{i-1} - \lambda_i$$
$$x_i = x_{i-1} + v_x \, dt$$
$$y_i = y_{i-1} + v_y \, dt$$

We did 100 iterations with dt = .1. The following shows the values for x and y every tenth iteration:

Time	x	y
1.0	0.109746	0.109746
2.0	0.307133	0.307133
3.0	0.473119	0.473119
4.0	0.511122	0.511122
5.0	0.506853	0.506853
6.0	0.50125	0.50125
7.0	0.49971	0.49971
8.0	0.499762	0.499762
9.0	0.499946	0.499946
10.0	0.500006	0.500006

An obvious extension is to allow more than one constraint per mass point. In that case, the energy function becomes:

$$\mathcal{E}_{\text{multiple}}(\mathbf{X}) = f(\mathbf{X}) + \Sigma_\alpha \lambda_\alpha g_\alpha(\mathbf{X}) \qquad (6)$$

Another extension, called augmented Lagrangian constraints (or ALCs) is formed by adding a penalty term to the basic differential equation (3). The penalty term corresponds to the energy:

$$\mathcal{E}_{penalty} = c/2(g(\mathbf{X}))^2 \; .$$

Thus, equation (3) becomes:

$$v_{x_i} = -\frac{\delta f}{\delta x_i} - \lambda \frac{\delta g}{\delta x_i} - cg \frac{\delta g}{\delta x_i} \; . \qquad (6)$$

Note that the value of the constant c need not be as large as for the penalty method alone.

Platt and Barr offer two alternative formulations of inequality constraints. One uses an extra slack variable, so if we want a constraint of the form $h(x) \geq 0$:

$$g(\mathbf{X}) = h(\mathbf{X}) - z^2 \qquad (7)$$

The other formulation uses a conditional:

$$g(\mathbf{X}) = \begin{cases} (h(\mathbf{X}))^2 & \text{if } h \leq 0 \\ 0 & \text{otherwise} \end{cases} \qquad (8)$$

We tried to use equation (8) in "Bouncer", our implementation of Witkin, Gleicher, and Welch's [5] constraint system, in order to prevent the baton from penetrating the floor. Here, $h(\mathbf{X}) = y$. This certainly prevented the baton from penetrating the floor, but because of the discontinuous nature of the constraint and the need for the constraint to be initially fulfilled in the Witkin system, the baton shot up with a velocity on the order of 10^{16}.

In general, the constraints outlined here are all fairly simple formulations. That is because there are so many variables involved in the finite element method that the constraints must be simple to make the entire problem tractable.

Julie R. Carrington
February 4, 1991

[5]

1. Witkin,A.; Gleicher,M.; Welch,W.; "Interactive Dynamics",Comp.Gr. Vol.24, #2, 3/90, pp11-22.

2. Witkin,A.; Welch,W.; "Fast Animation and Control of Nonrigid Structures", Comp.Gr. Vol.24, #4, 8/90, pp243-250.

The most important ideas from the first paper have already been detailed in J. Burg's "Constraint-Based Programming: A Survey". A few additional issues should be addressed such as advantages and disadvantages of this system and we will say a few words about our implementation, especially "Bouncer".

The second paper discusses in more detail the handling of non-rigid bodies. Witkin also reviews his system of constrained dynamics and discusses motion control in animation. I will just touch on a few points from (2).

'Bouncer' is a baton made of two spheres connected by a constraint keeping the distance between them constant. Visually the constraint is represented by a bar. The constraint ensures that no matter what forces are applied to one or both spheres, the bar gets no shorter or longer. For example, giving a 'kick' to one sphere creates both torque and linear acceleration in the system. In 'Bouncer', we include a floor in the model and drop the baton onto the floor under gravity. The problem is not that the constraints do not work but rather how to determine the amount of force we would expect the impact to give to the baton.

Initially, our hope was that we could treat the colliding ball in isolation and allow the constraints to provide the appropriate force to the other ball. The problem is that the amount of the impact force depends on the velocity and position of the second ball. We tried giving the colliding ball the amount of force needed so that if it were unencumbered by the second ball, it would continue with the same velocity ın the x direction but opposite in the y direction. In the case where the baton was falling vertically, that was the right force to precisely cancel the downward velocity of the second ball and the system halted.

All other combinations of applying force to one ball or both balls depending on the position and velocity of one ball or both balls result in either a gain or loss of energy over time.

One possible problem that we noted was because the force due to the constraints is calculated after the force due to collision is calculated. Since there is nothing in the constraint system specifying that the baton cannot penetrate the floor, after the force is calculated, the constraints can still push a ball through the floor.

For this reason, we tried to make the constraint system aware of the floor by using either one of the inequality constraints or the reaction constraints from [4]. The reaction constraints could prevent the baton from penetrating the floor but the problem of how much force to apply to force the baton to bounce without energy gain or loss remained. The inequality constraints (see equation (8) in the summary of [4]) were of no use because of the need for the constraints to be initially met in Witkin's system.

There are at least three possible explanations for the remaining problem that the system either gains or loses energy. One is that we simply haven't hit upon the right formula which takes all of the variables into account. That is the simplest but least likely explanation. Another possibility, suggested by [3] (1) on page 186, is that rigid body mechanics cannot adequately model this situation because the amount of force applied to the ball by the floor depends on the time that the ball is in contact with the floor which in turn depends on how long it takes the impact to propagate through the baton. The last possibility, suggested by Clay Johnson, is that the situation can be modelled by rigid body mechanics but the time step needs to be variable enough to catch the exact moment of impact in order that the ball never be allowed to penetrate.

The unique characteristic of this system is the it can be made interactive. Constraints can be dynamically added and deleted from the system as long as at the moment the (e.g.) attachment constraint is added, the constrained bodies are in fact attached. The most important disadvantage of this system is the fact that constraints must be initially fulfilled. As we saw, this was a problem when we wanted force the 'floor' constraint to become active only when the baton penetrated the floor. I believe the system is computationally expensive especially for flexible models. He uses only the simplest two dimensional flexible body as an example. Both of these problems are addressed in (2).

The second paper (2) reviews this same constraint system but adds a new twist. Here Witkin and Welch address the problem of computational complexity and suggest that by a very different method which works well in conjunction with their constraint system they can get similar results to those of Pentland [3].

In the finite element method, a body is modeled by a mesh of mass points. When the body is deformed, the forces on each mass point must be calculated with respect to all the other mass points. In Witkin's formulation, we still have the mass points but we consider them to be sitting in some region of space. We parameterize this region of space and deform the entire region. Then the motion of each mass point need only be calculated with respect to the global deformation rather than to each of the other mass points. This is as if we had a two dimensional figure lying on a bicubic patch. Changing the shape of the patch would change the shape of the figure.

The paper also addresses the problem of constraints having to be initially fulfulled. Having to have the position and velocity of the body match the constraint normally prohibits starting up a new constraint in the middle of an animation. He suggests that we can dynamically compute a spline curve segment to smoothly bring the body into the required initial state.

This appears to be true in the context of controlled motion, where the trajectory is defined and forces are calculated to move the body along that trajectory. This is the inverse dynamics approach. In this context also Witkin discusses impulses. Our hope initially was that this section on impulses would be of some help in 'Bouncer'. 'Bouncer' is modeled with a forward dynamics approach in which the forces are applied and new velocities and positions are determined by the system. Determining the trajectory the baton would be expected to follow after impact would surely be no easier than determining the forces which result from the impact.

Julie Carrington
February 4, 1991

[6] Witkin,A.;Kass, M., "Spacetime Constraints", Comp.Gr. Vol. 22, #4, 8/88.

Witkin and Kass suggest that a serious problem in using physical modeling methods for animation is that while motion can be made more realistic, the animator loses some control. While the animator is interested in the end result of the motion and the trajectory followed to achieve that end, simulation methods solve initial value problems. In other words, the motion is determined by the initial state of the system and by forces applied along the way.

The problem is illustrated by Pixar's *Luxo, Jr.*, (film, 1986). Originally, this was done using traditional keyframing where the computer interpolated between frames. Using a dynamic simulation approach, there are two possibilities. First, the animator could specify the motion of the way the lamp squats and pushes off to achieve a leap into the air and the system could calculate the motion which results from that force. That would be a long process of trial and error to get the proper amount of force for the desired trajectory. Second, we could specify where we want the lamp to jump and the desired trajectory as constraints and the lamp would follow that trajectory. Since the trajectory is produced by a constraint force, we still would have to manually determine a realistic squat and push off. What we need is a third method which would help us with the central problem of determining what the trajectory should be and how the character achieves that trajectory.

The most important difference between Witkin and Kass's constraint method and other dynamic simulation approaches is that the motion and forces are determined over the entire time interval rather than being determined sequentially at each time increment. For this reason, the name *spacetime constraints* was chosen.

We will follow the same simple example used in the paper to illustrate the method. In this example, we have a particle propelled by a jet engine which we want to move from point ***a*** to point ***b*** in a fixed amount of time using as little fuel as possible. We let the position of the particle be represented as a function of time, $\mathbf{X}(t)$, and the jet force be $\mathbf{f}(t)$. The mass of the particle is a constant m and g is the acceleration due to gravity. The equation of motion is:

$$m\mathbf{X}'' - \mathbf{f} - mg = 0 \qquad (1)$$

In addition, we need constraints:

$$c_0 = \mathbf{X}_0 - \boldsymbol{a} = 0 \qquad (2)$$
$$c_n = \mathbf{X}_n - \boldsymbol{b} = 0$$

and, assuming that our rate of fuel consumption is a simple function of the jet force such as $|\mathbf{f}|^2$, we want to minimize:

$$R = \int_{t_0}^{t_1} |f|^2 \, dt \qquad (3)$$

Thus, we want to find a force over the interval (t_0, t_1) such that the boundary constraints are satisfied and R is minimized. So far this looks very like other methods we have studied and we would expect to proceed across the interval, solving the initial value problem at each iteration. Instead, we are going to look at each of the functions, X(t) and f(t) independently where the equation of motion acts as a constraint relating the two functions. In addition, because we want to look at the entire interval, we must discretize the functions X(t) and f(t) by representing them as a sequence of values X_i and f_i, $0 \le i \le n$ with a step value of h between samples. The first and second derivatives of X(t) are approximated by:

$$\mathbf{X}_i' = \frac{\mathbf{X}_i - \mathbf{X}_{i-1}}{h} \qquad (4)$$

$$\mathbf{X}_i'' = \frac{\mathbf{X}_{i+1} - 2\mathbf{X}_i + \mathbf{X}_{i-1}}{h^2} \qquad (5)$$

Next we can substitute our approximation of X_i'' into the equation of motion (1) to get:

$$\mathbf{p}_i = m\frac{\mathbf{X}_{i+1} - 2\mathbf{X}_i + \mathbf{X}_{i-1}}{h^2} - \mathbf{f}_i - mg = 0, \quad 0 \le i < n \qquad (6)$$

The objective function R must also be discretized and so becomes:

$$R = h \sum_i |f_i|^2, \ 0 < i < n \tag{7}$$

Suppose that we want our particle to move from point **a** to point **b** in five iterations. Suppose in addition that we working in only two dimensions, x and y. Then we have a collection of independent variables, $\mathbf{S}_j = [x_j, y_j, fx_j, fy_j]$, $0 \le j \le 5$. Thus in our case, we have a vector with 24 independent variables, $[x_0, y_0, fx_0, fy_0, x_1, ..., fy_5]$. We also have a set of constraint functions $\mathbf{C}_i(\mathbf{S}_j)$, $1 \le i \le m$, consisting of both the given constraints and the **p**'s. In our case then we have:

$$C_0 = \mathbf{X}_0 - \mathbf{a} = 0$$

$$C_1 = m\frac{\mathbf{X}_0 - 2\mathbf{X}_1 + \mathbf{X}_2}{h^2} - \mathbf{f}_1 - \mathbf{m}\mathbf{g} = 0$$

$$C_2 = m\frac{\mathbf{X}_1 - 2\mathbf{X}_2 + \mathbf{X}_3}{h^2} - \mathbf{f}_2 - \mathbf{m}\mathbf{g} = 0$$

$$C_3 = m\frac{\mathbf{X}_2 - 2\mathbf{X}_3 + \mathbf{X}_4}{h^2} - \mathbf{f}_3 - \mathbf{m}\mathbf{g} = 0$$

$$C_4 = m\frac{\mathbf{X}_3 - 2\mathbf{X}_4 + \mathbf{X}_5}{h^2} - \mathbf{f}_4 - \mathbf{m}\mathbf{g} = 0$$

$$C_5 = \mathbf{X}_5 - \mathbf{b} = 0$$

The basic idea of the solution method is to solve two linear systems in sequence. In the first solution, we get an approximation *to the* $\mathbf{S}_j$ *which minimizes* R without regard to the constraint equations, $\mathbf{C}_i$. This approximation is called $\hat{\mathbf{S}}_j$. Next we project $\hat{\mathbf{S}}_j$ onto the null space of the Jacobian matrix of the constraint equations at the same time as driving them to zero. This results in another approximation called $\tilde{\mathbf{S}}_j$. Finally, $\Delta\mathbf{S}_j = \hat{\mathbf{S}}_j + \tilde{\mathbf{S}}_j$. We iterate until $\mathbf{C}_i = 0$ and we can no longer decrease R without violating the constraints.

The first linear system we solve is:

$$-\frac{\delta R}{\delta S_i} = \sum_j \frac{\delta^2 R}{\delta S_i \delta S_j} \hat{S}_j$$

and the second is:

$$- Ci = \sum_j \frac{\delta C_i}{\delta S_j} (\hat{S}_j \quad + \quad \tilde{S}_j.) \ .$$

Values for each of these variables are given in the paper for our example but I believe they are incorrect. We are told that the Jacobian matrix should contain:

$$\frac{\delta C_i}{\delta X_j} = 2m/h^2, \ i = j$$

$$= -m/h^2, \ i = \pm \ i$$

$$= 0 \text{ otherwise.}$$

$$\frac{\delta C_i}{\delta f_j} = 1, \ i = j$$

$$= 0 \text{ otherwise.}$$

It seems that all of these signs of reversed. In addition, the values of the Jacobian and Hessian matrices are given as:

$$\frac{\delta R}{\delta f_i} = 2f_i \quad \text{and}$$

$$-\frac{\delta^2 R}{\delta S_i \delta S_j} = 2, \qquad i = j$$

$$= 0 \text{ otherwise.}$$

They should be:

$$\frac{\delta R}{\delta f_i} = 2hf_i \quad \text{and}$$

$$-\frac{\delta^2 R}{\delta S_i \delta S_j} = 2h, \qquad i = j$$

$$= 0 \text{ otherwise.}$$

The choice of linear system solvers is critical. The matrices are large but sparse and often (as in this example) not square. They suggest using a pseudo-inverse in connection with a sparse conjugate gradient algorithm. We are referred to [14] in their references for details on this algorithm but we are told that it is $O(n^2)$ for typical problems. I assume that n is the number of independent variables; in our case 24.

The balance of the paper describes the particular example of Luxo using spacetime constraints. The lamp is assumed to be made of rigid bodies connected by joint equipped with spring-like muscles. Constraints are used to specify initial and final poses and to specify that Luxo be on the floor at the beginning and end of the jump. Just as in our example, the other constraints are the physics constraints, p_i. The objective function specifies that we want to minimize power to the muscles. As figure 3 on page 166 of the paper shows, after several iterations, Luxo executes a realistic jump including the needed squash-and-stretch to get off of the ground and follow-through.

The goal in this system is not merely to force an object to follow a predefined path by the use of constraints but to help the animator in defining a realistic path to follow. The goal seems to be met to a miraculous extent. For example, figure 7 on page 167 shows Luxo doing a ski jump in what looks like perfect form.

To: M. Moshell

From: Julie Carrington

Date: November 25, 1991

Re: Project number VSL91.2, Constrained Dynamics

Excerpted from "Constraint-Based Programming: A Survey" by

Jennifer Burg, Charles Hughes, J. Michael Moshell, Sheau-Dong Lang

Physically-based modeling, a growing area of graphics research, interprets constraints as the basis for forces or energies that can be used to move objects in a physically-realistic manner. Using techniques borrowed from optimization theory and classical mechanics, a physically-based modeling system converts constraint equations into constraint forces or represents constraints as energy fields which effectively pull the object in the desired direction. Thus physically-based modeling places constraints in a world governed by Newtonian mechanics.

Physically-based modeling begins with the building of an object from primitive parts. Geometric constraints effectively snap the parts together. For example, a constraint can form a joint between two parts of a body (similar to ThingLab's *merge*, which joins two points). Joint limits can also be expressed through constraints. Once the object has been pieced together, constraints can be used to position or animate it. For example, a constraint can anchor a point to a location in space, or it can require a point to follow a predefined path [Barzel and Barr 1988].

The two most commonly-reported constraint methods for physically-based modeling are reviewed in the next two sections. These methods have been adapted and augmented by a number of researchers, as discussed in Section 3.

1. Penalty Method

The penalty method is a technique borrowed from optimization theory. The basic optimization problem is to find a vector $\overline{x}$ which locally optimizes (minimizes or maximizes) a function $f(\overline{x})$. The basic constrained optimization procedure optimizes the function subject to a given constraint. For example, we may wish to

minimize $f(x_1, x_2) = 2x_1^2 + 3x_2^2$

subject to $g(x_1,x_2) = x_1 + 3x_2 - 10 = 0$,

where $f(x_1,x_2)$ is the objective function and $g(x_1,x_2) = 0$ is the constraint.

The penalty method looks for an approximate solution to this problem by first defining a new objective function $h(\overline{x})$ which includes a penalty for violating the constraint. Since we want the penalty to indicate how far we are from satisfying the constraint, it is reasonable that the penalty should be of the form $k[g(\overline{x})]^2$. Now the problem is to minimize

$$h(\overline{x}) = f(\overline{x}) + k[g(\overline{x})]^2.$$

The non-negative constant k is a weighting factor which indicates how strongly we insist on the constraint. When $k = 0$, the constraint is ignored. As $k \to \infty$, the constraint is completely satisfied. Figure 1 illustrates how $h(\overline{x})$ approaches the constrained minimum for increasing values of k.

The penalty formulation of a minimization problem can be used to model the constrained motion of an object. Here, the objective function $U(\overline{x})$ represents the potential energy of the unconstrained physical system, while the penalty function $k[g(\overline{x})]^2$ acts like a spring with its own potential energy, attempting to pull the object in the direction of the constraint:

$$E(\overline{x}) = U(\overline{x}) + k[g(\overline{x})]^2 \qquad [1]$$

The penalty method is also easily generalized to multiple constraints since the constraint terms can be summed. The problem

minimize $U(\overline{x})$, subject to $g_\alpha(\overline{x}) = 0;\ \alpha = 1, 2, \ldots, n$

becomes

$$\text{minimize } E(\overline{x}) = U(\overline{x}) + \sum_{\alpha=1}^{n} k_\alpha[g_\alpha(\overline{x})]^2 \qquad [2]$$

Since the object will minimize its potential energy over time, we can now solve the constrained motion problem as a minimization problem. The simplest optimization algorithm is a numerical procedure called *gradient ascent/descent*. (We will put aside our energy function for the moment and illustrate the gradient ascent (*hillclimbing*) procedure with a two-dimensional function since this is easier to depict graphically. We will then return to solve the energy function for our physical model.)

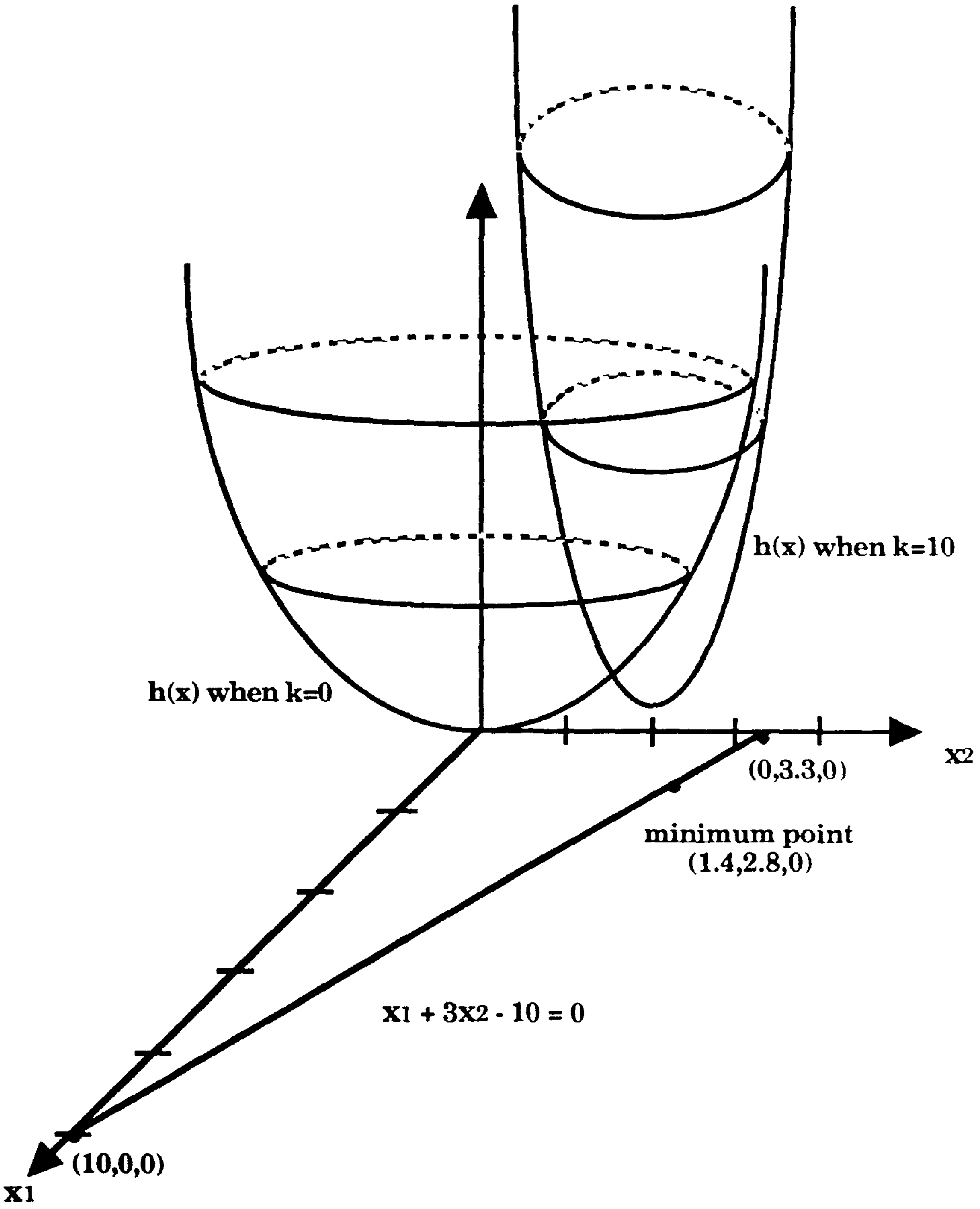


Figure 1. Penalty method approaches constrained minimum

Consider the graph in Figure 2 representing a function

$y = f(x_1,x_2)$

which we wish to maximize. The problem is to start at some initial position *(x_1,x_2)* and to find a path *S* which will lead to a local maximum for *y*. However, we don't want to take a circuitous path: we want to go straight up the hill. The hillclimbing method is based on the observation that the rate of steepest ascent will be along the gradient of *f(x_1,x_2)*. (Analogously, the rate of steepest descent will be opposite the gradient. See [Gottfried and Weisman 1973] for the derivation.)

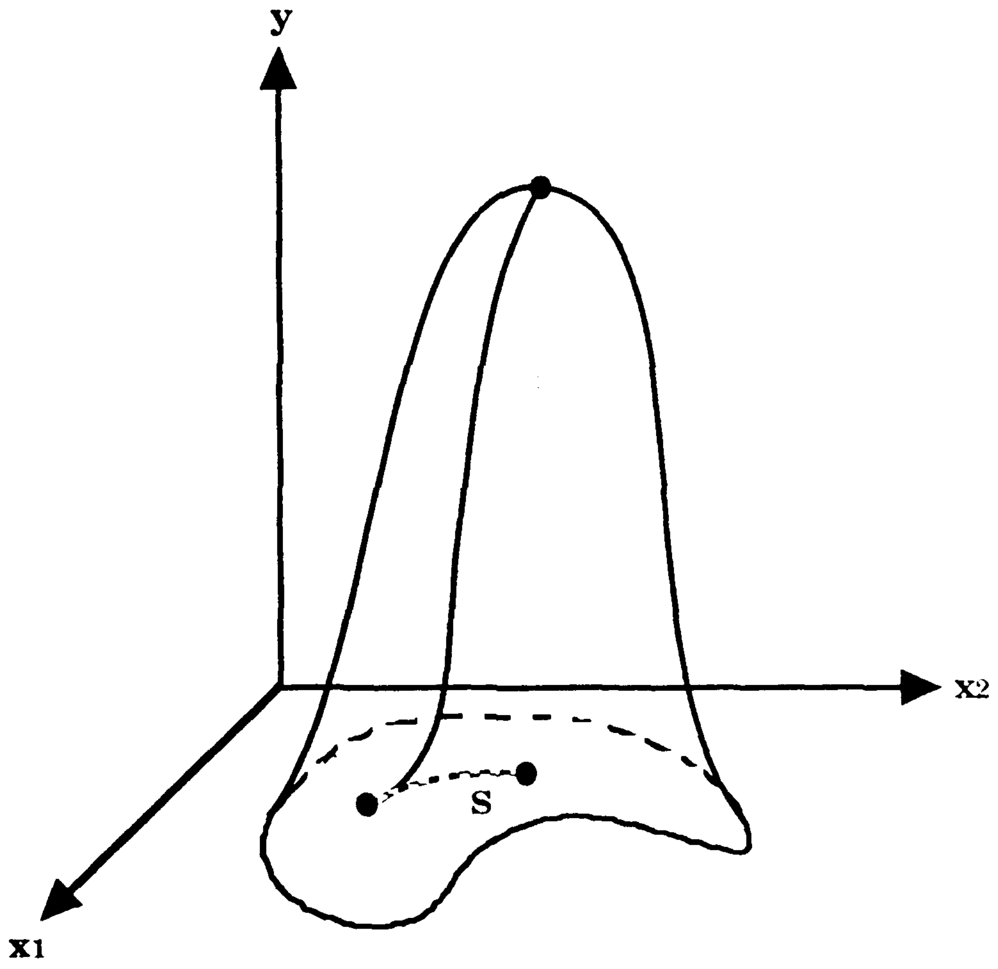


Figure 2. Hill climbing to maximum

We can understand intuitively why the hillclimbing method works if we picture the contour lines of the objective function projected down to the x_1, x_2 plane (Figure 3). (The contour lines represent the contours of the function when y is set to different constants.) Saying that we want to move in the direction of the gradient of the function is equivalent to saying that we want our path S always to run perpendicular to these contour lines. This will take us directly up the "hill" as the values of x_1 and x_2 vary. Thus, we have:

$$\frac{dx_i}{dt} = \frac{\partial f}{\partial x_i} \quad \text{for } i = 1,2.$$

(For steepest descent, the variables move in the opposite direction of the gradient: $\frac{dx_i}{dt} = -\frac{\partial f}{\partial x_i}$).

We can now use Euler's method to solve the differential equations. Let x_{ik} denote the value of independent variable x_i at time t_k. To get the value of the independent variables at time t_{k+1}, we use:

$$x_{ik+1} = x_{ik} + h\frac{\partial f}{\partial x_i} \,,$$

where h is the step size. We continue to step up the hill in this manner until the gradient becomes sufficiently small to indicate that we have arrived at a maximum.

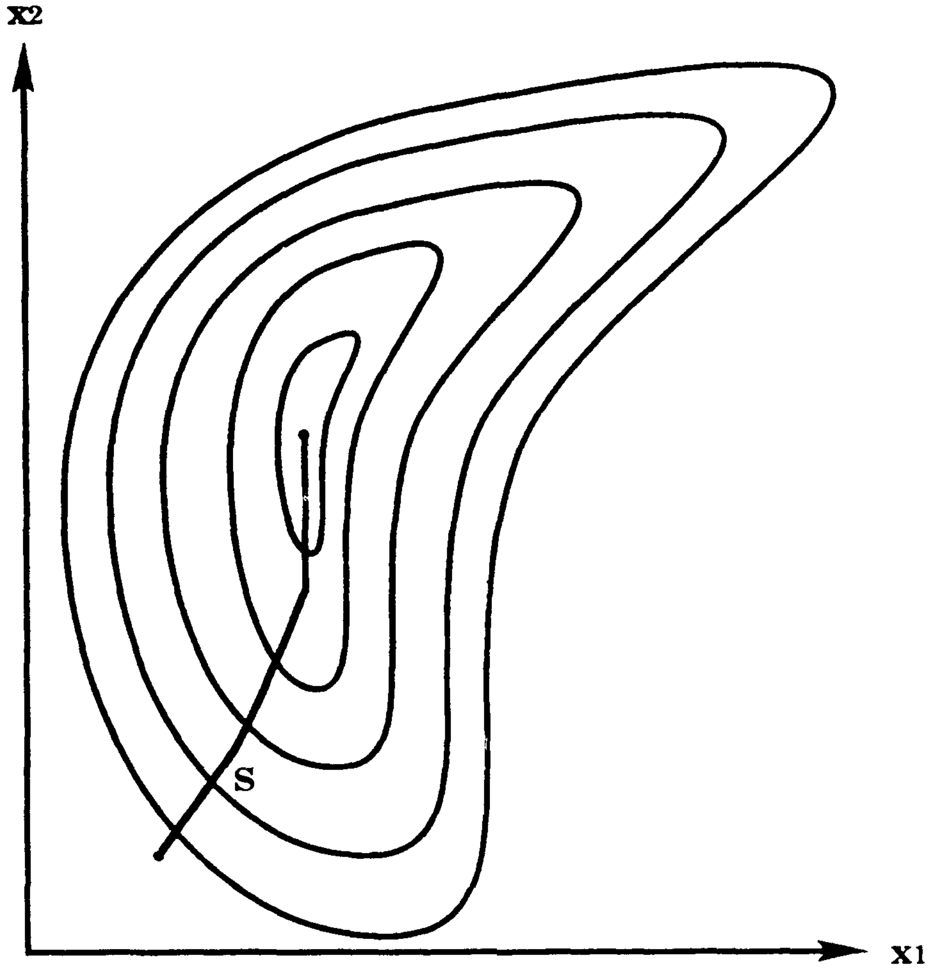


Figure 3. Projection of contour lines onto x1-x2 plane

As noted above, the penalty method can be used to minimize the potential energy of a physical object which we wish to place in motion. The objective function to be minimized is given by equation [1]. Applying the gradient descent method, we move the independent variables in $\overline{x}$ in a direction opposite to the gradient of the function. This yields:

$$\Sigma_j M_{ij} \frac{dx_j}{dt} = F(\overline{x}) - 2kg(\overline{x}) \frac{\partial g}{\partial x_i} \qquad [3]$$

where M_{ij} is the generalized mass matrix and $F(\overline{x})$ is the generalized force on the system [Platt 1989].

A simple example (Figure 4) will illustrate this method. Let (x_1,x_2), (x_3,x_4) denote the positions of two unit mass balls in 2-d space. Their initial positions are (0.4,0) and (0.5,0), respectively. The balls are constrained by "springs" to the (0,1) and (1,1) positions, respectively, and they are also attached to each other by a spring.

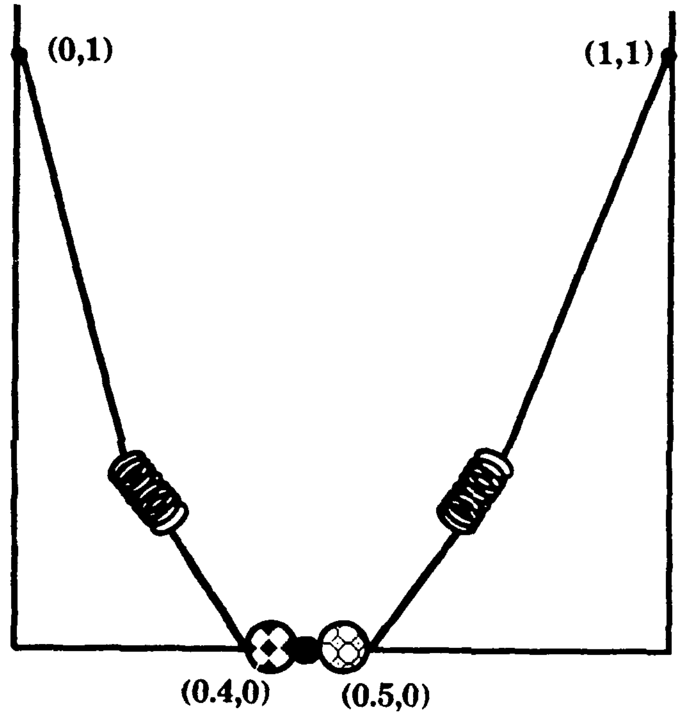


Figure 4. Penalty method simulating balls on springs

This gives the following constraints:

$$g_1(\overline{x}) = \sqrt{x_1{}^2 + (x_2-1)^2}\ ,$$

$$g_2(\overline{x}) = \sqrt{(x_3-1)^2 + (x_4-1)^2}\ ;\ \text{and}$$

$$g_3(\overline{x}) = \sqrt{(x_1 - x_3)^2 + (x_2 - x_4)^2}\ .$$

The balls are also constrained not to go below the ground. To achieve this effect, we activate a penalty force only when a ball moves to a negative y position, as follows:

$g_4(\overline{x}) = u_1 x_2$ where $u_1 = 1$ when $x_2 < 0$; else $u_1 = 0$;

$g_5(\overline{x}) = u_2 x_4$ where $u_2 = 1$ when $x_4 < 0$; else $u_2 = 0$.

Giving each constraint $g_\alpha(\overline{x})$ a weight of k_α, we get a penalty term $\sum_{\alpha=1}^{n} k_\alpha [g_\alpha(\overline{x})]^2$ as follows:

$$k_1[x_1{}^2 + (x_2-1)^2] + k_2[(x_3-1)^2 + (x_4-1)^2] + k_3[(x_1-x_3)^2 + (x_2-x_4)^2] + k_4 u_1 x_2{}^2 + k_5 u_2 x_4{}^2$$

To apply the gradient descent method, we use equation [3]. Since the balls are unit mass particles, the mass matrix is the identity and falls out of the equation. The external force $F(\overline{x})$ consists only of gravity, and thus the force vector is $F(\overline{x}) = [0,-9.8,0,-9.8]$. Thus, equation [3] yields:

$$\frac{dx_i}{dt} = F(\overline{x})_i - \sum_{\alpha=1}^{n} 2k_\alpha g_\alpha(\overline{x}) \frac{\partial g_\alpha}{\partial x_i}, \quad i = 1,2,3,4$$

where

$$\sum_{a=1}^{n} 2k_{a\cdot,}a(\overline{x}) \frac{\partial g_a}{\partial x_1} = 2k_1 x_1 + 2k_3(x_1-x_3);$$

$$\sum_{a=1}^{n} 2k_a g_a(\overline{x}) \frac{\partial g_a}{\partial x_2} = 2k_1(x_2-1) + 2k_3(x_2-x_4) + 2k_4 u_1 x_2;$$

$$\sum_{\alpha=1}^{n} 2k_{\alpha} g_{\alpha}(\overline{x})\frac{\partial g_{\alpha}}{\partial x_3} = 2k_2(x_3-1) - 2k_3(x_1-x_3); \text{ and}$$

$$\sum_{\alpha=1}^{n} 2k_{\alpha} g_{\alpha}(\overline{x})\frac{\partial g_{\alpha}}{\partial x_4} = 2k_2(x_4-1) - 2k_3(x_2-x_4) + 2k_5 u_2 x_4.$$

Integrating these with a time step of 0.01 and then sampling every tenth point results in motion of the balls characterized by the time and positions in Table 1.

Time	x1	x2	x3	x4
0.0	0.40	0.00	0.50	0.00
0.1	0.20	0.66	0.37	0.58
0.2	0.16	0.37	0.31	0.16
0.3	0.15	0.28	0.30	0.02
0.4	0.15	0.26	0.29	-0.01
0.5	0.14	0.28	0.29	0.02
0.6	0.14	0.29	0.29	0.05
0.7	0.14	0.31	0.29	0.10
0.8	0.14	0.26	0.29	0.00
0.9	0.14	0.26	0.29	0.00
1.0	0.14	0.28	0.29	0.03

Table 1. Positions of balls in penalty method

While the penalty method is fairly simple to use, it does not satisfy constraints exactly. This is an advantage in that a compromise among constraints is sometimes desirable, but a disadvantage in that an object held together by multiple "springs" may look too loosely connected. Another disadvantage is that as the constraint weights increase, the differential equations become *stiff* due to the widely separated time constants. Most numerical methods require time steps on the order of the fastest time constant. However, such large time steps may cause the "springs" to bounce unrealistically.

Application of the penalty method in physically-based modeling is discussed in [Witkin, Fleischer, and Barr 1987], [Platt and Barr 1988], and [Platt 1989].

2. Inverse Dynamics

In some applications, it is necessary that constraints be fulfilled exactly. This can be accomplished by a process of *inverse dynamics*. The forward dynamics problem entails computing an object's behavior given the forces which act on it. The inverse dynamics problem is just the opposite: Given the constraint equations which define an object's structure, location, and behavior, we compute *constraint forces* which cause the object to move in an appropriate manner. A constraint force works like an invisible hand which guides an object in the correct direction or prevents it from moving beyond its limits no matter what other external forces are exerted on the object.

Witkin, Gleicher, and Welch [1989] present a derivation of the physical equation which yields the appropriate constraint force. Given in the problem are a set of constraint functions $c_i(q, t)$ and a vector q of the object's independent variables, and a time t. Each constraint function c_i depends on the state of the independent variables and possibly also on time. We say that the constraint is satisfied when $c_i(q, t) = 0$.

The goal is to find a constraining force C which, when added to the known external force Q, will result in motion which is consistent with the constraints. The object will be moved in accordance with Newton's second law of motion, which in generalized form is:

$$M_{ij}\ddot{q}_j = C_j + Q_j. \qquad [1]$$

M is the generalized mass matrix of the object being moved.[1] C is the vector of unknown constraint forces, and Q is the vector of known external forces with respect to each independent variable. The constraint force effectively cancels any component of Q which would cause the object to violate its constraints. Once C is known, it can be added to Q. Then $\ddot{q}$, the second time derivative of the independent variables, can be determined. It is then possible to integrate the differential equation over time and move the object.

This equation cannot be solved directly since both C and $\ddot{q}$ are unknown. We need more information. Since we want the constraints always to be satisfied, we know that each constraint c_i must be 0 at initial time t_0, the rate of change of c_i must be 0, and that rate of change must not change from 0. Thus we have $\ddot{c}_i = 0, 1 \leq i \leq n$ (where n is the number of constraints).

Finding $\ddot{c}_i$ and substituting into $\ddot{q}_j = W_{jk} (C_k + Q_k)$, we get

$$\frac{\partial c_i}{\partial q_j} W_{jk} (C_k + Q_k) + \frac{\partial \dot{c}_i}{\partial q_j} \dot{q}_j + \frac{d^2 c_i}{dt^2} = 0 \qquad [2]$$

where W is the inverse of matrix M.

[1] According to the summation convention, the appearance of an index twice in a term implies summation. Thus $M_{ij}q_j$ is equivalent to $\Sigma_j M_{ij}q_j$, i.e., row i of matrix M times vector q.

Rearranging, we get:

$$-\frac{\partial c_i}{\partial q_j} W_{jk} C_k = \frac{\partial c_i}{\partial q_j} W_{jk} Q_k + \frac{\partial \dot{c}_i}{\partial q_j} \dot{q}_j + \frac{d^2 c_i}{dt^2} \qquad [3]$$

which is a matrix representation of a system of linear equations with the force vector C unknown. Noting that $\frac{\partial c_i}{\partial q_j}$ is an $n \times m$ matrix and W is an $m \times m$ matrix (where n is the number of constraint equations and m is the number of independent variables), we can see that equation [3] represents n equations and m unknowns. Since in general $n < m$, we have fewer equations than unknowns. We still need more information.

The fact that there are fewer equations than unknowns indicates that the system is underconstrained, which is as it should be. If the system were completely constrained, nothing could move. In an underconstrained system such as this, there exist many values for the constraint force C which would satisfy the equation. However, we want to add only enough force to cancel out any component of Q which would cause the object to deviate from the constraints.

To satisfy the constraints, the object can move only along the tangent planes to the surfaces $c_i = 0$. Thus, a constraint force that lies along the gradient to the constraints c will cancel any illegal force while not adding or deleting energy from the system. This observation yields:

$$C_j = \lambda_i \frac{\partial c_i}{\partial qj}, \qquad [4]$$

where λ is vector of scalars. The components of λ are known as *Lagrange multipliers*, and this technique of inverse dynamics is sometimes referred to as *the Lagrange multiplier method.* (See [Witkin, Gleischer, and Welch 1989] for a discussion of *the principle of virtual work.*)

We now have:

$$-[\frac{\partial c_i}{\partial q_j} W_{jk} \frac{\partial c_r}{\partial q_k}] \lambda_r = \frac{\partial c_i}{\partial q_j} W_{jk} Q_k + \frac{\partial \dot{c}_i}{\partial q_j} \dot{q}_j + \frac{d^2 c_i}{dt^2} \qquad [5]$$

Note $\frac{\partial c_i}{\partial q_j}$ is the Jacobian matrix for the constraint equations. (That is, each row i in the matrix represents that gradient vector for constraint c_i.) W_{jk} is the inverse of the mass matrix, and $\frac{\partial c_r}{\partial q_k}$ is the transpose of the Jacobian matrix. This gives us, on the left-hand side of equation [5], an $n \times m$ matrix multiplied by an $m \times m$ multiplied by an $m \times n$, yielding an $n \times n$ matrix. λ_r is an $n \times 1$ vector of unknowns. On the right-hand side, we get an $n \times 1$ vector of known values. We can now solve this system of linear equations for λ_r. Once λ_r is known, we can get C from equation [4], and we can finally solve for $\ddot{q}$.

Theoretically, the above solution should supply a constraint force sufficient to ensure that the objects always maintain their constraints. However, due to errors introduced in discretizing the integration, it becomes necessary to include a "feedback" term which inhibits drift. Thus the total force becomes:

$$Q_j + C_j + \alpha c_i \frac{\partial c_i}{\partial q_j} + \beta c_i \frac{\partial \dot{c}_i}{\partial q_j},$$

where α and β are constants.

Application of this method can be illustrated with a simple example. We will create a "midpointline" like the one constructed in

ThingLab, with another fixed-length line attached at an endpoint. We will apply an initial force to the unattached endpoint of the midpointline and watch it move in response to the force.

Let $q = [q_1\ q_2\ q_3\ q_4\ q_5\ q_6\ q_7\ q_8]$ be the vector of independent variables with initial values [0 0 10 0 20 0 20 10]. That is, we have four points (q_1,q_2), (q_3,q_4), (q_5,q_6), (q_7,q_8) at initial positions (0,0), (10,0), (20,0), and (20,10), respectively. The first two constraints make the first three points collinear and equidistant. The third constraint fixes the length of the line between (q_5,q_6) and (q_7,q_8) at 10 units.

$$c_1\text{:}\ 2q_3 - q_1 - q_5 = 0$$
$$c_2\text{:}\ 2q_4 - q_2 - q_6 = 0$$
$$c3\text{:}\ 100 - (q_5 - q_7)^2 - (q_6 - q_8)^2 = 0$$

The points are particles of unit mass. Thus the mass matrix is the identity matrix and can be dropped out of the equation. We will exert an initial force of 10 units in both the x and y directions on point $(q1,q2)$, which gives an initial force Q = [10 10 0 0 0 0 0 0].

The constraint Jacobian matrix is

$$\begin{bmatrix} -1 & 0 & 2 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 2 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2(q_5-q_7) & -2(q_6-q_8) & 2(q_5-q_7) & 2(q_6-q_8) \end{bmatrix}$$

At the initial moment t_0, the second term on the right-hand side of equation [5], $\frac{\partial \dot{c}_i}{\partial q_j} \dot{q}_j$, will be 0 since the initial velocity is 0. For time $t > t_0$, we have $\frac{\partial \dot{c}_i}{\partial q_j} \dot{q}_j = \frac{\partial}{\partial q_j}(\frac{d}{dt}(c_i(q_r(t)))) \dot{q}_j = \frac{\partial}{\partial q_j}(\frac{\partial c_i}{\partial q_r} \dot{q}_r) \dot{q}_j$. The third term on the right-hand side, $\frac{d^2 c_i}{dt^2}$, will always be 0 in this example since the constraint functions do not depend on time.

Using a time increment of 0.1, we see the following motion of the object. (Time *t* = 1 occurs after 10 time slices.)

t	q 1	q 2	q 3	q 4	q 5	q 6	q 7	q 8
0	0.00	0.00	10.00	0.00	20.00	0.00	20.00	10.00
1	83.00	82.00	10.33	36.00	19.83	-0.09	20.00	9.91
2	1.67	1.64	10.67	73.00	19.67	-0.18	20.00	9.82
3	2.50	2.45	11.00	1.09	19.50	-0.27	20.00	9.72
4	3.33	3.27	11.33	1.46	19.33	-0.35	20.00	9.63
5	4.17	4.09	11.67	1.82	19.17	-0.44	20.00	9.53
6	5.00	4.91	12.00	2.19	19.00	-0.52	20.00	9.42
7	5.83	5.72	12.33	2.55	18.83	-0.60	20.00	9.33
8	6.67	6.54	12.67	2.92	18.67	-0.69	20.00	9.23
9	7.50	7.36	13.00	3.29	18.50	-0.77	20.00	9.12
10	8.33	8.17	13.33	3.66	18.34	-0.84	20.00	9.02

Table 2. Positions of midpointline in Lagrangian method

Figure 5 shows the initial and final positions of this midpoint example. The force on the first point has caused it to move in a 45° angle, upward and to the right. This is consistent with the fact that the force was equal and positive in the *x* and *y* directions. This force, and the constraint that the second line maintain a fixed length, has caused the midpoint line to compress from a length of 20 to a length that is slightly less than 14, whereas the fixed length line retains its length of 10.

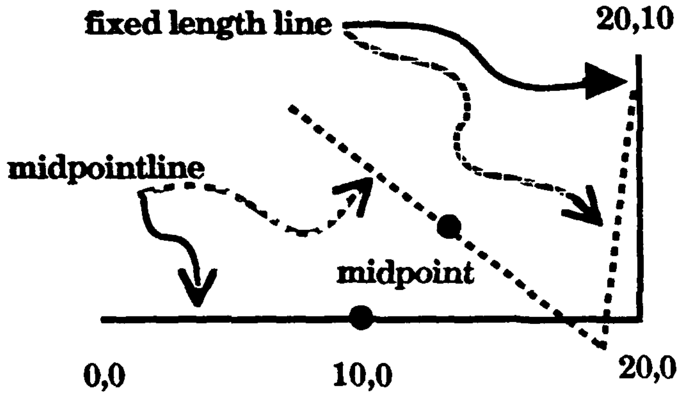


Figure 5. Lagrangian approach to midpoint with attached fixed length line. Solid lines are at t=0, dotted are at *t=10.*

3. More Physically-Based Constraints

Isaacs and Cohen [1987] combine behavior functions, kinematic constraints, and inverse dynamics to control the motion of jointed figures. Behavior *functions determine higher* level goals of motion, like a hand reaching for an object or a car stopping to avoid a cliff. Kinematic constraints specify exactly where a part of an object is to go. Inverse dynamics techniques then determine the forces which would result in the goal of motion.

Barzel and Barr [1988] also use inverse dynamics on rigid bodies. They divide the modeling problem into two parts: moving the parts of an object so that the object satisfies an initially unmet constraint (causing the object to "self-assemble"); and maintaining the constraint as the object moves and interacts with other objects. A catalog of useful constraints and an explicit algorithm for computing the constraint forces are given.

Platt and Barr [1988] apply reaction constraints and augmented Lagrangian constraints to flexible models, that is, putty-like objects. Reaction constraints are used to model the collision of a flexible model with a polygonal model. Augmented Lagrangian constraints combine the penalty method and the Lagrange multiplier method.

Platt [1989] applies the constraint methods to neural networks. The neural networks are actually differential equations that can be solved using standard techniques from optimization theory and numerical analysis, or implemented directly as circuits. Platt also reviews the constraint methods applied to physically-based modeling.

ACKNOWLEDGEMENTS

This work was supported in part by the Florida High Technology and Industry Council, and the U.S. Army's Program Manager for Training Devices. We thank Dr. Robert Brigham for his helpful editorial comments. We also thank Julie Carrington and Xin Li for their help in implementing the physical modeling examples.

REFERENCES

AGHA, G. 1985. Actors: a model of concurrent computation in distributed systems. PhD Thesis. Ann Arbor, Mi. University of Michigan.

BARZEL, R., AND BARR, A. 1988. A modeling system based on dynamic constraints. *Computer Graphics* 22, 4 (August), 179-187.

BERINGER, H., AND PORCHER, F. 1989. A relevant scheme for Prolog extensions: CLP(Conceptual Theory). *6th International Conference on Logic Programming* (Lisbon, Portugal, June 19-23). pp. 131-148.

BORNING, A. 1986. Defining constraints graphically. *Human Factors in Computing Systems CHI'86 Conference Proceedings* (April 13-17), pp. 137-143.

BORNING, A. 1981. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Trans. Program. Lang. and Syst.* 3, 2 (Oct.), 353-387.

BORNING, A. 1979. ThingLab -- a constraint-oriented simulation laboratory. Ph.D. dissertation, Computer Science Dept., Stanford Univ., Stanford, Calif.

BORNING, ET AL. 1987. Constraint hierarchies. *OOSPLA '87 Proceedings* (Orlando, Fl., October 4-8, pp. 48-60.

BORNING, A., ET AL. 1989. Constraint hierarchies and logic programming. *6th International Conference on Logic Programming* (Lisbon, Portugal, June 19-23), pp. 149-164.

CHANG, CHIN-LIANG, AND LEE, RICHARD CHAR-TUNG. 1973. *Symbolic Logic and Mechanical Theorem-Proving*. Academic Press, New York.

CLARK, K. AND GREGORY, S. 1986. Parlog: parallel programming in logic. *ACM Trans. Program. Lang. and Syst.* 8, 1 (Jan.), 1-49.

CLINGER, W. L. 1981. Foundations of Actor semantics. PhD thesis, MIT, Cambridge, Mass.

COHEN, J. 1990. Constraint logic programming languages. *Commun. ACM* 33, 7 (July), 52-68.

COLMERAUER, A. 1990. An introduction to Prolog III. *Commun. ACM* 33, 7 (July), 69-90.

COOPER, P. 1988. Structure recognition by connectionist relaxation: formal analysis. *Proceedings of the 7th Biennial Conference of the Canadian Society for Computational Studies of Intelligence. CSCSI '88* (Edmonton, Alberta, Canada, June 6-8), pp. 148-155.

DAVIS, E. 1987. Constraint propagation with interval labels. *Artificial Intell.* 32, 3 (July), 281-331.

DAVIS, L. S., AND ROSENFELD, A. 1981. Cooperating processes for low-level vision: a survey. *Artificial Intell.* 17, 1-3 (August), 245-263.

DECHTER, R. 1988. Constraint processing incorporating backjumping, learning, and cutset-decomposition. *IEEE 4th Conf. on AI Applications* (San Diego, Calif., March 14-18), pp. 312-317.

DECHTER, R. 1986. Learning while searching in constraint-satisfaction-problems. *Proceedings AAAI 86, Vol. 1* (Philadelphia, Pa., Aug. 11-15), pp. 178-183.

DECHTER, R., AND DECHTER A. 1987. Belief maintenance in dynamic constraint networks. *Proceedings AAAI* 87, *Vol. 1* (Seattle, Wash., July 13-17), pp 37-42.

DECHTER, R., AND PEARL, J. 1988. Network-based heuristics for constraint satisfaction problems. *Artificial Intellig*. 34, 1 (Dec.), 1-38.

DESCOTTE, Y., AND LATOMBE, J.-C. 1985. Making compromises among antagonist constraints in a planner. *Artificial Intellig*. 27, 2 (Nov.), 185-217.

DOYLE, J. 1979. A truth maintenance system. *Artificial Intellig*. 12, 3 (Nov.), 231-272.

DUISBERG, R. A. 1986. Constraint-based animation: temporal constraints in the Animus system. Tektronix Laboratories Technical Report No. CR-86-37, August 1986.

EGE, R. K. 1989. Direct manipulation user interfaces based on constraints. *IEEE* 1989, 374-380.

EGE, R. K., MAIER, D., AND BORNING, A. 1987. The filter browser: defining interfaces graphically. *Proceedings of the European Conference on Object-Oriented Programming* (Paris, France, June), pp. 155-165.

FIKES, R. E., AND NILSSON, N. J. 1971. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intellig*. 2 (1971), 189-205.

FREEMAN-BENSON, B. N., MALONEY, J., AND BORNING, A. 1990. An incremental constraint solver. *Commun. ACM* 33, 1(Jan.), 54-63.

FREUDER, E. C. 1978. Synthesizing constraint expressions. *Commun. ACM* 21, 11 (Nov.1978), 958-965.

GASHNIG, J. 1974. A constraint satisfaction method for inference making. *Proceedings of the 12th Allerton Conf. On Circuit and System Theory* (Urbana, Ill., Oct. 2-4), pp. 866-875.

GASHNIG, J. 1978. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing-assignment problems. *Proceedings of the 2nd National Conference of the Canadian Society for Computational Studies of Intelligence*. CSCSI '78 (Toronto, Ont., July 19-21), pp. 268-277.

GASHNIG, J. 1977. A general backtrack algorithm that eliminates most redundant tests. *Proceedings of the 5th International Joint Conference On Artificial Intelligence.* IJCAI (Cambridge, Mass., Aug. 22-25), p. 457.

GOLDBERG, A., AND ROBSON, D. 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Mass.

GOLDSTEIN, H. 1965. *Classical Mechanics.* Addison-Wesley, Reading, Mass.

GOTTFRIED, B. S., AND WEISMAN, J. 1973. *Introduction to Optimization Theory.* Prentice-Hall, Englewood Cliffs, N. J.

GRAF, T. 1987. Extending constraint handling in logic programming to rational arithmetic. European Computer-Industry Research Centre (ECRC) Internal Report, Munich, Germany, 1987.

GROSSMAN, M., AND EGE, R. K. 1987. Logical composition of object-oriented interfaces. *OOPSLA '87 Proceedings* (Orlando, Fl., October 4-8), pp. 295-304.

HARALICK, R., ET AL. 1978. Reduction operations for constraint satisfaction. *Inform. Sci.* 14 (1978), 199-219.

HARALICK, R. M., AND ELLIOTT, G. L. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intellig.* 14, 3 (Oct.), 263-313.

HARALICK, R. M., AND SHAPIRO, L. G. 1979. The consistent labeling problem: part I. *IEEE Trans. Pattern Anal. Machine Intell. PAMI-1,* 2 (April), 173-184.

HAYES-ROTH, B., ET AL. 1986. Protean: deriving protein structure from constraints. *Proceedings AAAI 86, Vol. 1* (Philadelphia, Pa., Aug. 11-15), pp. 904-909.

HEINTZE, N., ET AL. 1987. *The CLP(R) Programmer's Manual,* Version 2.0. Dept. of Computer Science, Monash University, Clayton 3168, Victoria, Australia, June 1987.

HESTENES, M. 1975. *Optimization Theory.* Wiley & Sons, New York.

HEWITT, C., AND BAKER, H. 1978. Actors and continuous functionals. In E. J. Neuhold, Ed. *Formal Descriptions of Programming Concepts*. North-Holland Publishing Co., New York, pp. 367-390.

HUMMEL, R. A., AND ZUCKER, S. W. 1983. On the foundations of relaxation labeling processes. *IEEE Trans. Pattern Anal. Machine Intell. PAMI-5*, 3 (May), 267-287.

ISAACS, P., AND COHEN, M. 1987. Controlling dynamic simulation with kinematic constraints, behavior functions, and inverse dynamics. *Computer Graphics* 21, 4 (July), 215-224.

JAFFAR, J., AND MICHAYLOV, S. 1987. Methodology and implementation of a constraint logic programming system. *Proceedings of the 4th International Conference on Logic Programming* (Melbourne, Australia, May), pp. 196-219.

JAFFAR, J. AND LASSEZ, J-L. 1987. Constraint logic programming. *Proceedings of 14th ACM Symposium on Principles of Programming Languages* (Munich, Germany, Jan.).

KASIF, SIMON. 1986. On the parallel complexity of some constraint satisfaction problems. *Proceedings AAAI 86, Vol. 1* (Philadelphia, Pa., Aug. 11-15), pp. 349-353.

KOWALSKI, R., AND KUEHNER, D. 1971. Resolution with selection function. *Artificial Intellig.* 3, 3 (1970), 227-260.

KNUTH, D. E. 1979. *TeX and METAFONT: New Directions for Typesetting*. American Mathematical Society and Digital Press, 1979.

LELER, W. 1988. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley, Reading, Mass.

LLOYD, J.W. 1987. *Foundations of Logic Programming*. 2nd ed. Springer-Verlag, New York.

LOVELAND, D. 1978. *Automated Theorem Proving: A Logical Basis*. North-Holland Publishing Company, New York.

MACKWORTH, A. 1977. Consistency in networks of relations. *Artificial Intellig.* 8, 1 (Feb.), 99-118.

MACKWORTH, A. K., AND FREUDER, E. C. 1985. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intellig.* 25, 1 (Jan.), 65-74.

MALONEY, J. H., BORNING, A., AND FREEMAN-BENSON, B. N. 1989. Constraint technology for user-interface construction in ThingLab. *OOPSLA '89 Proceedings* (New Orleans, La., October 1-6), pp. 381-388.

MARON, M. J. 1982. *Numerical Analysis: A Practical Approach.* Macmillan, New York.

MINSKY, M. 1975. A framework for representing knowledge. In *The Psychology of Computer Vision*, P. H. Winston, Ed. McGraw-Hill, New York, pp. 211-277.

MOHR, R., AND HENDERSON, T. C. 1986. Arc and path consistency revisited. *Artificial Intellig.* 28, 2 (March), 225-233.

MONTANARI, U. 1974. Networks of constraints: fundamental properties and applications to picture processing. *Inform. Sci.* 7 (1974), 95-132.

NELSON, G. 1985. JUNO, a constraint-based graphics system. *Computer Graphics*, (July 1985), 235-243.

NEWELL, A., AND SIMON, H. 1972. *Human Problem Solving*. Prentice-Hall, Englewood Cliffs, N. J.

NUDEL, B. 1983. Consistent-labeling problems and their algorithms: expected complexities and theory-based heuristics. *Artificial Intellig.*, 21, 1-2 (March), 135-178.

O'DONNELL, M. J. 1985. *Equational Logic as a Programming Language.* MIT Press, Cambridge, Mass.

PLATT, J. 1989. Constraint methods for neural networks and computer graphics. PhD Thesis, California Institute of Technology.

PLATT, J., AND BARR, A. 1988. Constraint methods for flexible models. *Computer Graphics* 22, 4 (August), 279-288.

ROBINSON, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of ACM* 12, 1 (Jan.), 23-41.

ROSENFELD, A., HUMMEL, R. A., AND ZUCKER, S. W. 1976. Scene labeling by relaxation operations. *IEEE Trans. Syst. Man Cybern. SMC-6*, (May), 420-433.

SARASWAT, V. A. 1989. Concurrent constraint programming languages. Ph.D. dissertation, Computer Science Dept., Carnegie Mellon Univ., Pittsburgh, Pa.

SHAMBLIN, J. E., AND STEVENS, J. 1974. *Operations Research: A Fundamental Approach.* McGraw-Hill, New York.

SHAPIRO, EHUD, ED. 1987. *Concurrent Prolog, Volume 1 and 2.* MIT Press, Cambridge, Mass.

SIMONIS, H., NGUYEN, H.N., AND DINCBAS, M. 1988. Verification of digital circuits using CHIP. *Proceedings of the IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification,* Glasgow, Scotland, July.

STEFIK, M. 1981. Planning with constraints (MOLGEN: Part 1). *Artificial Intellig.* 16, 2 (May), 111-140.

SUSSMAN, G., AND STEELE, G. 1980. CONSTRAINTS--a language for expressing almost-hierarchical descriptions. *Artificial Intellig.* 14, 1 (Aug.), 1-39.

SUTHERLAND, I. 1963. Sketchpad: a man-machine graphical communication system. MIT Lincoln Laboratory Technical Report No. 296, Jan. 30, 1963.

SWAIN, M. J., AND COOPER, P. R. 1987. Parallel hardware for constraint satisfaction. *Proceedings AAAI* 87, *Vol. 1* (Seattle, Wash., July 13-17), pp. 682-686.

VAN HENTENRYCK, P. 1989a. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Mass.

VAN HENTENRYCK, P. 1989b. Parallel constraint satisfaction in logic programming. *Proceedings of the 6th International Conference on Logic Programming* (Lisbon, Portugal, June 19-23), pp. 165-180.

VAN WYK, C. J. 1981. IDEAL user's manual,. Bell Labs CS TR No. 103 (Dec.).

WALINSKY, C. 1989. CLP(Σ*): constraint logic programming with regular sets. *6th International Conference on Logic Programming*, (Lisbon, Portugal, June 19-23), pp. 180-196.

WALTZ, D. 1975. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, P. H. Winston, Ed. McGraw-Hill, New York.

WITKIN, A., FLEISCHER, K., AND BARR, A. 1987. Energy constraints on parameterized models. *Computer Graphics* 21, 4 (July), 225-229.

WITKIN, A., GLEICHER, M., AND WELCH, W. 1990. Interactive Dynamics. *Computer Graphics*, 24, 2 (March), 11-22.

Appendix G:

The PM Physical Modeling System

Dynamic Terrain Project
1990-91

Object Oriented Physical Modeling (the PM System)

Curtis Lisle, Dr. J. Michael Moshell
Institute for Simulation and Training
University of Central Florida, Orlando, FL

August 1990

VSL Memo 90.19

Introduction

Physical modeling can be defined as a modeling technique in which various physical parameters are incorporated into an animated model; allowing the behavior of the model to be simulated according to known natural laws [Barzel 89]. With this technique, all animated objects are simply forced to obey several physical laws instead of requiring the animator to specify every detail of the motion.

This project involves applying the concepts of Object Oriented Programming to the area of physical modeling. In the PM system, physical objects are modeled by objects in Smalltalk. The interaction between the objects is carried out by the exchange of Smalltalk messages.

Section One: Previous Work

There are several physical modeling efforts currently in progress by different research groups in both universities and industry. This chapter briefly surveys some of the more important work done by these groups.

1.1 Penalty Method

The *penalty method* approach to physical modeling has been pioneered primarily by Alan Barr and his associates at Cal-Tech, see [Barr 89], [Barzel 89], and [Platt 89]. This approach considers a simulation as a group of objects each acting independently. "Penalties" are inflicted on the objects if their behavior does not conform to the desired behavior. Objects are, in effect, forced to behave appropriately.

Penalties are inflicted on objects primarily by exerting forces. The farther an object is away from the desired behavior, the stronger the *penalty force* exerted on it. Figure 1.1 shows a case where a penalty force is correcting the incorrect interpenetration of two objects. The ball should have bounced instead of penetrating the table, so the farther the interpenetration, the bigger the penalty force. A math model can be applied here: considering the penalty force as a spring which pulls harder the farther it is stretched. This has the desired effect of correcting the incorrect situation, however, it is computationally expensive in practice.

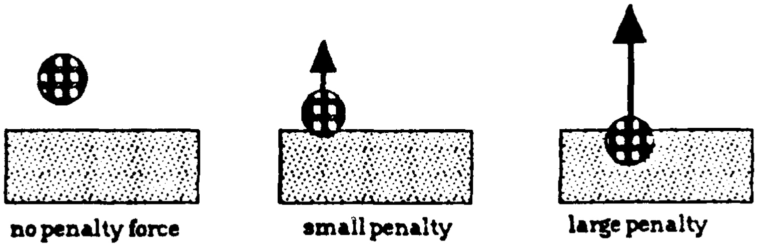


Figure 1.1 - Penalty correcting interpenetration

Springs exert conservative forces on objects causing no loss of energy. This is physically correct, but leads to perceptible oscillation unless damping is applied. Since the penalty force will act on an object in a discontinuous manner, the damping values and the best time interval at which to computer the simulation are difficult to determine. This can cause a need to "tweak" each simulation until it looks right.

A positive feature of the penalty method is that a wide variety of constraints can be represented using penalty forces. Consider the mechanical situation of a ball connected to the end of a stick with a short, strong string. The spring, which exerts a penalty force, models the condition that the objects are fastened together. This idea can be extended: building complex machines using penalty forces to hold objects in position. This type of "rubber band machine" is, however, susceptible to the same oscillation problems described above. In its complete implementation the following process is performed:

1. All constraints built into a scenario exert initially unknown penalty forces.

2. An energy equation is constructed for each object representing the object's state and the penalties exerted on it.

3. An energy minimization approach is applied to find the new state values of the object.

1.2 DYNAMO

In [Isaacs 87], the authors describe DYNAMO, a system for computing dynamic motion of linked bodies. This system performs *inverse dynamics* to calculate the positions and orientations of each body in a linked system. *Inverse dynamics takes the current state and finds the forces necessary to perform a desired behavior.*

The DYNAMO system contains an equation solver to perform the inverse dynamics calculations and integrators to derive the new state from the previous state. An *adaptive time interval* is used to maximize the simulation throughput while limiting the maximum error allowed into the calculations. After every time cycle, the object states are examined for accrued error and, if necessary, the time slice is adjusted.

Constraints and damping terms are applied directly on each object's DOFs (degrees of freedom) to make the motion look more realistic. For example, the motion of an elbow joint is constrained below a maximum angle of bending. DYNAMO also supports *behavior functions* like "follow a path" or "reach for an object" which exert themselves on the objects with forces.

1.3 Modeling Object Collisions

The approaches described in [Moore 88] and [Baraff 89] are both notable efforts in the area of detecting and resolving collisions between objects. Each of them have created an analytical method for finding points of collision of arbitrary shaped objects at any position and orientation.

The goal of the method in [Moore 88] is to prevent interpenetration of bodies by finding the collision point and exerting a calculated force vector at the collision point. The Cyrus-Beck clipping approach [Rogers 85] is extended to produce an *inside-outside test* for a vertex and polygon. Given a point in space, the test will determine which side of a polygon the point lies on. If a point lies on the "inside" of every polygon of an object, then it interpenetrates the object. For collision detection, the test is applied between every vertex of one object and every polygon in the other object. This is an $O(m^2n^2)$ algorithm (with m vertices in the first object and n faces in the second object).

Two methods of collision resolution are presented in [Moore 88]. The first involves applying a spring force on interpenetrating vertices; essentially the same as the penalty method. The second approach is analytical, creating a set of fifteen equations and fifteen unknowns which completely specify the motion of both objects after a collision [MacMillan 36]. The analytical approach has the advantage that it only needs to be applied once for every collision. The spring approach often must be calculated repeatedly over a large number of small time intervals. Finally, [Moore 88] extends the analytical approach to hierarchical articulated figures (multi-joint structures).

The work outlined in [Baraff 89] draws a distinction between *colliding contact* and *resting contact.* Colliding contact is impulsive in nature, varying discontinuously over time. Resting contact, like a normal force holding an object on a table, is continuous over time. The author finds a set of contact points where forces can be exerted to maintain correct resting contact. A linear programming approach is developed to solve for resting forces using a heuristic, and this is extended to cases where multiple-objects are in resting contact with each other.

In [Baraff 89], the resting forces are calculated by satisfying constraints placed on the objects in contact; interpenetration and friction are handled in this manner. This analytical approach is compared with *propogational approaches* which handle only two-object-at-a-time collisions.

1.4 Lagrangian Approach

The physical modeling team at Carnegie Mellon, lead by Andrew Witkin [Witkin 89 #1] [Witkin 89 #2] [Witkin 90], have taken a *conservation of energy approach* where constraints and forces all contribute to the energy levels in objects. In [Witkin 89 #2], a catalog of useful constraints is presented which are expressed in terms of the energy contribution to the object. The sum of constraints applied to an object must do *virtual work*: it must change the object's motion, but not violate the laws of conservation of energy by giving or taking energy away from the object.

According to [Witkin 89 #1], physical modeling becomes a problem of *constrained optimization.* Several physically accurate solutions to motion may exist, but one way will most accurately satisfy all the constraints placed on an object. Animation-like motion where objects stretch, lean, and squash has resulted from expressing these motions in constraints and allowing the system to calculate all the motion.

A Lagrangian-based solution is developed and presented in [Witkin 90] which does not suffer from the oscillation and small time slice problems found with the penalty method.

Section Two: The Object Oriented Approach

Object oriented programming (OOP) is an approach to programming which has gained acceptance quickly in both the academic and industrial communities. Using OOP methods generally results in a shorter design cycle and a more extendable, more flexible final software product.

In OOP, the data structures and routines which use these data structures are bound together into an *abstract data type* (ADT) [Meyer 88]. Once the ADT is finished, the only access to the data structure is through the interface provided by the routines (called *methods*) bound in the ADT. This is referred to as *encapsulation* and is an important feature of OOP. Software written using this approach is constructed by designing the relevant ADT's, called *classes* in object oriented terminology, creating *instances* (or specific copies of the AD1 s) for use in the program, and letting these instances (called *objects*) communicate with each other by sending *messages* to each other - accomplishing the goal of the system as their end result.

Object Oriented Design in the PM System

The uniqueness of this project is due primarily to the application of the object oriented methodology to the design and creation of a physical modeling system. This is in contrast to the previous systems described which use a more classical software design approach.

As this paper will describe, the PM system must perform the same types of mathematical calculations done by other systems. However, the way this is accomplished through the interaction of objects makes this system unique.

The PM system is written in Smalltalk-80 and resides on Sun SparcStations under version 2.5, the Sun 386i under version 2.4, and the Apple Mac-IIx under version 2.4. It has been interfaced to two different animation packages to render the output of the physical simulations.

2.1 The Controller, Objects, and Constraints

Since the simulation is represented entirely through the interaction of a number of objects, a number of different classes of objects were created. The following sections briefly describe the three classes of objects necessary to carry on a physical simulation in the PM system.

2.1.1 The Simulation Controller

An instance of the simulation controller class (referred to as "the controller" from now on) manages all the other object instances involved in the simulation. The controller instance is invoked directly by the user, receives some setup messages, and then executes the simulation a step at a time.

Other simulation objects are maintained in lists by the controller - segregated by object type. During each time step, the controller sends messages, as appropriate, to each object in the lists - causing them to interact with each other and cause the simulation.

The five lists managed by the controller are the *objList* (of physical objects), the *preConstList* (of preconstraint objects), the *postConstList* (of postconstraint objects), the *connectorList* (of connector objects), and the *monitorList* (of monitor objects). A diagram of this is shown in Figure 2.1. The connections on the diagram underneath the lists represent simulation messages. These messages are sent between senario objects during the simulation.

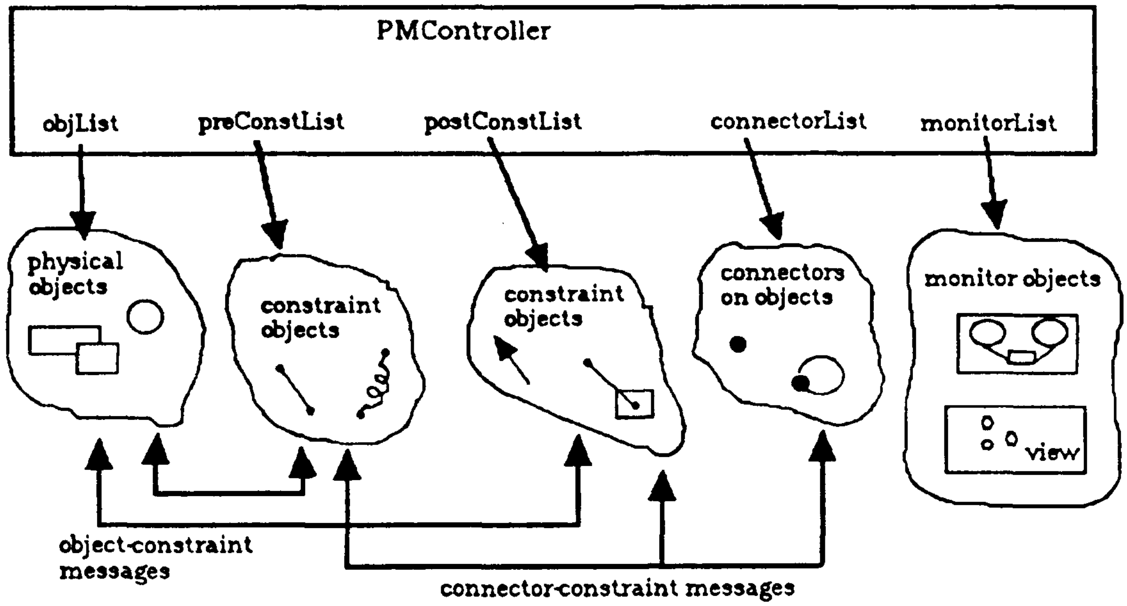


Figure 2.1 - Structure of a PM System Simulation

2.1.2 The Physical Objects

Each physical object in a scenario (ball, table, box, etc) is represented by a software object instance placed in the *ObjList.*

During each time cycle, the controller asks the objects if they have collided with each other. An assumption the PM system was designed around can be stated as follows: *Since physical objects know their state more completely than any other object in the simulation, they should make the decision whether they have collided with another object by communicating directly with that object.*

The same assumption was made about handling *collision resolution* (exchanging momentum) between objects. This is accomplished by direct communication between the objects involved instead of by going through the controller or allowing the controller to decide what happens when a collision occurs.

Accurately maintaining internal state requires that an object correctly respond to external influences (constraints, collisions). Each object must, therefore, be able to receive forces and exert forces on other objects in the scenario in a manner analogous to what occurs in the physical world.

2.1.3 The Constraint Objects

These objects perform the role of influencing the motion of one or more physical objects by "pushing or pulling" on them. Several different classes of constraints exist in the PM system, and a constraint of a particular type is applied by instantiating an object of this class and attaching it to a physical object.

Constraints usually implement a rule such as "stay on a wire between two points" or "pull with a force proportional to the position of the object". They can also be viewed as math models for devices in the physical world such as pins or springs. Interesting motion is usually accomplished by allowing several constraints to act on a physical object simultaneously or in succession. The exact types of constraints available in the PM system will be covered in Chapter Six.

2.1.4 The Monitor Objects

Any object maintained by the controller which just passively watches the scenario is considered a monitor. It may generate a graphical window showing the positions of objects; it may write out the scenario results in a manner which can replayed at a later time; or it may perform any other function which can be performed based on information available in the object lists maintained by the controller.

The first two functions described have already been implemented as monitors. This feature of the controller can be considered as an opening for future expansion and interfacing of the PM system to other systems. One example of an additional monitor application useful in a distributed simulation could send messages to shadow objects existing on a machine different than the controller. Through this, the controller could send state updates to the shadow objects.

2.2 Class Hierarchy in the PM System

The PM system was the first major object oriented (OO) design effort undertaken by the author. Several notable results were obtained from the OO approach. Some of these are outlined in the following sections of this chapter.

2.2.1 Features Provided by the Hierarchy

Usually, as an important software engineering project is undertaken, any modules which successfully work are saved, copied, and then modified to avoid the danger of "breaking" something which worked before the modifications. This form of versioning is successful, but tends to produce a collection of different instances of the same module which all perform slightly different functions – a dangerous situation.

During the development of the PM system, the rule "Don't adapt, descend" from [Ezzel 90] has been loosely followed. The PM system consists of a number of classes each of which have more ability than the parents from which they inherited. Descendents are usually more capable than their parent classes; for example, the class hierarchy *Object* -> *PMObject* -> *PMSphere* -> *PMSphereWithForces* -> *PMSphereWithAngular*. This is a cleaner situation than having a collection of "similar" methods. The ability to descend into subclasses helped manage the software development process considerably.

In general, the divisions of classes in the hierarchy represent divisions in the type of object (different object typess are located in different branches of the hierarchy). See the chart in Appendix A for a complete class hierarchy. The class hierarchy is a good record of the directions taken during the PM system development.

2.2.2 Division of Labor

Another design goal followed during the PM system development was to maintain a *division of labor* between the objects in a simulation. As much as possible, each object would have a clearly defined purpose and subclasses of each other would have similar purposes. For example, objects from any constraint classes will act to constrain the behavior of a physical object(s), and physical object classes will never perform this type of function (no classes like *PMCubeOnASpring*).

Design decisions were made to support the division of labor goal. Some of these are listed below:

1. Physical objects will detect collisions

2. Physical objects will resolve collisions
3. The controller will manage time, but not generate displays
4. Monitors will create displays, reports, etc.
5. Constraints will be isolated from geometry when possible

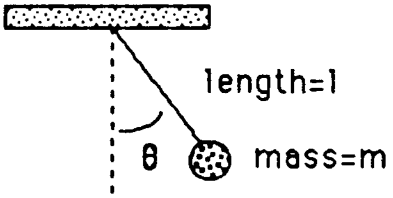
2.3 Class Interaction

The problem of "Who should handle .... ?" is closely related to the division of labor problem. Any problems which exist in physically modeling a simple environment must be solved by one of the object instances. Which object should handle a given problem?

Whether the objects or the controller detect collisions and whether the constraints or connectors (see Chapter Six) handle the geometry are both examples of this problem.

Section Three: Time Slicing in the PM System

When the motion of a particular physical system is studied, a set of equations are usually developed which describe the position of the objects for this particular system. The simple pendulum, along with the usual approximate equations for the angular velocity and period, is a good example. Figure 3.1 shows this case as described in [Ohanian 89], which holds true for small values of θ :



$$\omega = \sqrt{g / l}$$

$$\text{period} = 2\pi / \omega = 2\pi\sqrt{l/g}$$

Figure 3.1 - Math model of a simple pendulum

Some physical modeling systems calculate the position and orientation of the objects modeled by solving the equations in the math model of each object. This type of system is limited because it requires a math model for every type of object interaction included in the simulation.

Instead of solving a set of explicit math models, the PM system uses forces to represent interaction. For the simple pendulum example, a constraint force is placed on a ball by a constraint object. The continuous force causes circular motion.

3.1 Time Slice Sub-cycles

In order to model the behavior of multiple objects simultaneously reacting to external forces, each unit of simulation time (a time slice) is divided into two sub-cycles. The definition of the sub-cycles is as follows:

> force exchange: (1st sub-cycle) During this sub-cycle, all objects which are interacting in any fashion because of a collision or an applied constraint will exert forces on each other. All objects sum the forces they receive, but do not react.
>
> object reaction: (2nd sub-cycle) All objects use the sum of forces and moments about their center of masses to update their velocities, positions, and orientations.

The separate sub-cycles were designed to eliminate any problems which may occur due to the order in which the objects are handled by the controller. Additional sub-cycles may be necessary to resolve collisions between multiple objects - objects must divide their momentum between incident colliders if more than one collider is present. This will be discussed later in Section Six.

3.2 Response time problem

The two sub-cycle approach is not without its problems. Take the case of the famous momentum toy shown in figure 3.2. Assume that ball B1 strikes ball B2 during time t1. Momentum conservation is done along the axis of intersection yielding forces on both ball B1 and B2 (during the *force exchange* sub-cycle of t1). Since B2 had no velocity previous to the collision, it will not exert any force on B3 during t1's force exchange period even though they are in contact. This means that after t1's *object reaction* sub-cycle, B2 has a velocity while B3 still does not. Therefore, during t2, *ball B2 will move into B3* . The ball B2 in the real toy would have, instead, transferred all its momentum to B3 which then passes it to B4, which passes it to B5.

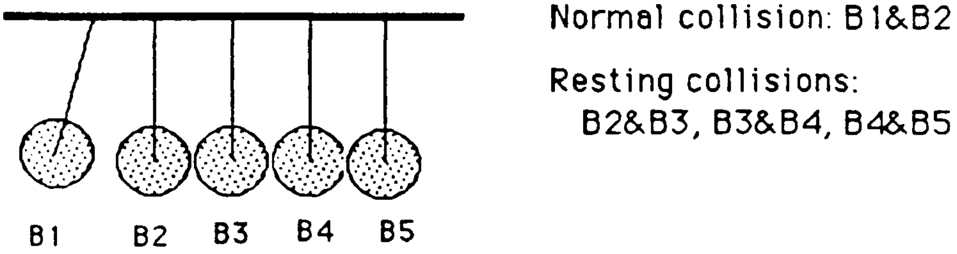


Figure 3.2 - Momentum toy

This problem can be summarized as follows: *The PM system transfers momentum incrementally between multiple objects in collision (one per time slice) whereas physical objects can transfer momentum almost instantly.* This is an artifact of imposing a simulation clock on events. The amount of error detected is related to the size of the time slice used. The actual physical toy can be considered as working with an infinitely small time slice.

3.4 Areas for Improvement

Accurate collision detection requires a very small time slice with the current approach. Unless a better collision detection algorithm is used by the objects in the simulation, collisions can be completely missed if the time slice is not sufficiently small.

If a collision detection algorithm can be developed which will predict possible collisions the time step before they occur (or before they are missed), a recursive simulation approach could be implemented as an extension to the current PM system.

Once an impending collision is detected, the objects which might collide could be instantiated into a separate simulation (like a recursive *sub-simulation*) and be allowed to interact with a smaller time interval. The sub-simulation would run to completion, return the accurate positions and velocities of the objects involved, and the top-level simulation could continue.

Section Four: Collisions

For any physical modeling system, collisions between objects are an important, if not a primary, problem. Two groups in particular, [Baraff 89] and [Moore 88], have developed fairly complete solutions for this problem. Although this was not the primary focus of this project, it was an area which required substantial effort. In the PM system, objects are responsible for deciding if they have collided with each other (see Section 2.3).

During each simulation step, the simulation controller queries each object to determine whether it has collided with any other of the objects – taking $O(n^2)$ time to determine all inter-object collisions. No effort to improve the efficiency of this algorithm was performed since time complexity was not a main issue of this project.

4.1 Detecting a Collision

Since the controller queries each physical object in the simulation about whether it has collided with every other physical object, each object must make a decision about whether a collision has occurred or not. Each object, which is either a block or sphere type, detects collisions through their *collidedWith: aPMObject* method. The algorithm is based on the distance between object centers and the size of the two objects.

4.1.1 size representation

To correctly detect collisions between objects, all objects must represent their individual size and know how to communicate this to other objects when requested. Size is communicated by giving every object a method which returns its size. This method is called by other objects whenever the size is needed.

If an object is a sphere, it returns its radius (from the objSize instance variable) when queried. However, the separate x, y, and z sizes of a block are represented by an additional instance variable in the *PMCube* class called 'size' the value of which is of type *PMVector*. Individual components of the size are retrieved by sending a message to the vector instance.

This raises a contradiction between the way that instances of the classes *PMCube* and *PMSphere* respond to the *size* message. Blocks will return a vector type while spheres return a scalar. This has been handled by providing a separate method for instances of *PMCube* called *sizeVec* which always returns a vector. The *size* method of a *PMCube* always returns a scalar value of zero. This will prevent any spheres from believing they have collided with a block when, in reality, they have only collided with the block's bounding sphere. Instead, the block will detect the collision because the sphere answers its *size* message correctly. This results in blocks always "discovering" they have collided with spheres at the right instant because only the block knows its actual size.

4.2 Collision Resolution:

Collisions are resolved in the PM system by calculating the transfer of momentum which occurs in every collision. If momentum transfer is always calculated correctly and the transfer is done by exerting forces, all physical objects will maintain the correct level of kinetic and potential energy as well momentum [Beer 88]. Energy conservation was not specifically considered except for the *handleCollisionWith:* method in the class *PMSphere* which was an early collision model derived from [Blinn 87]. All later collision resolution methods dealt strictly with momentum transfer.

4.3 Areas for Improvement

Most of the object classes in the PM system currently detect and resolve collisions in the global (or world) coordinate system. It makes much more intuitive sense for each object to perform calculations and make its decisions with respect to its own local coordinate system. The rewriting of the algorithms to local coordinates which was done for the class *PMCubeWithLocal* should be done for all objects.

Collision detection decisions are currently based on object position only. This is too dependent on the size of the time cycle since object positions are calculated only at discrete points in time. A predictive detectng algorithm was described from the perspective of the time slice at the end of Chapter Four.

The algorithmic approach used in [Baraff 89] or [asdfsdf] would be a more robust approach to collision detection. An algorithm could be implemented in a Smalltalk object and referenced by the controller or by physical objects. This would probably best be done during the controller's *findAllCollisions* method where it tells objects to handle their own collisions. The new approach in Smalltalk would look like:

```
|solver|
((obj1 collidedWith: obj2) or: [(obj2 collidedWith: obj1)])
        ifTrue: [
              solver = CollisionResolver  new.
              solver  resolveCollisionBetween: obj1 and: obj2].
```

This would not require any changes to the physical object classes.

Section Five: Constraints in the PM System

To simulate any "interesting" motion, for example, motion where objects follow a path, don't cross a border, or stay a fixed distance from each other, a physical modeling system must provide a mechanism to externally affect the behavior of any or all objects in a simulation. This has been accomplished through the use of *constraints* placed on an object or on its motion [Barzel 89]. As discussed earlier, the PM system implements constraints by having constraint objects exert external forces on physical objects – causing the objects to react in a desired manner. This is consistent with the method of applying constraints used in [Barzel 89], [Witkin 90] where all constraints are calculated and then expressed as external forces on the objects.

A design goal of the PM system was to have constraints interact with objects in a manner as consistent as possible with object-to-object interaction. As implemented in the PM system, constraint objects can be considered as specific math models contained in Smalltalk objects. The design, invocation, and interaction of these models is described in the following sections.

5.1 Constraints as Objects

Since each physical object in a simulation is represented by a *PMObject* subclass instance, it is consistent with this paradigm to represent any constraints as instances of similar subclasses of *PMConstObj*. These constraint objects are also maintained by the controller in a manner similar to the physical objects.

When instantiated, a constraint object is *attached* to one or more physical objects upon which it will act during the course of the simulation. During each time slice, the controller activates the constraint: instructing it to apply itself to the physical objects it is connected to through a force the constraint object calculates. The constraint object will then execute its own method describing how the force is calculated. An example is shown in Figure 6.1 where an instance of the *PMConstObj* subclass *PMAnchoredSpring* is attached between a fixed point in space and the object – an instance of

PMSphereWithForces . During instantiation, the spring is given the location of its base point (the anchor) and a pointer to the object its free end is connected to. When the spring is invoked by the controller, it queries the object in order to calculate the force it will apply. Depending on the type of constraint, the needed information will vary, but as a rule: *Constraint objects find the information they need to apply themselves by querying the objects they are attached to.* In the case of the spring, it asks for the sphere's position so it can use the equation F=k(freeEndPosition-basePointPosition) to calculate its applied force.

5.2 Including Constraints in the PM System

Several methods were considered before deciding on a constraint approach for the PM system. The primary problem was ensuring that constraints and physical objects interacted together correctly. Since the controller manages all of the object interaction, it must be able to handle the constraint objects along with the physical objects in the simulation. Several approaches for the controller were considered:

1. separate constraint list - All constraint objects would be inserted in a list analogous to the *objList*, but designed for constraint objects only. A method similar to the *findAllCollisions* would traverse the constraint list and apply the constraints. Here constraints are treated as an additional paradigm for the objects, separate from collisions.

2. constraint objects in objList - The list of objects in a scenario would include both physical objects and constraint objects. A problem arises since the constraint could "collide" with an object it is connected to (or even an object it is not connected to) and be a slave in the collision. This could be solved by sorting the object list (so constraints come first), or by affecting the way a constraint object responds to a *collidedWith:* message. With either method, constraints must always be collision masters if the possibility of colliding with a physical object exists.

Since the constraint objects are not instances from a *PMObject* subclass and have different purposes, it was decided that the first option using separate lists was a cleaner and more intuitive approach. This approach was implemented in the PM system.

5.3 PMControllerWithConstraints

The design of the controller is affected by the addition of constraint objects to simulations. Therefore, a subclass of *PMController* called *PMControllerWithConstraints* was created. *PMControllerWithConstraints* maintains the constraint objects in a list separate from the physical objects in a scenario. In addition to calling the *advanceTime* methods for each physical object every time slice, the controller calls the *apply* methods for every constraint in its constraint list. This necessitates a uniform constraint-to-controller interface consisting of the same methods to interact with the controller even if the purpose of the constraint is different.

It will be seen in the next sections that two different classes of constraints exist. The controller in the PM system maintains a separate list for each of these constraint classes (so it has three lists: one for objects, two for constraints). These two constraint classes are referred to as *preconstraints* and *postconstraints.*

5.4 Preconstraints

The first of the two major classes of constraints is called preconstraints. *A preconstraint is defined as a constraint whose force calculation is not dependent on other forces already incident on the object* . The spring is a good preconstraint example since it does not need to know what forces have been applied on the object already. To be an accurate spring model, it will pull with a force proportional to the amount it is stretched. Therefore, it is dependent solely on the object's position. A list of example preconstraints is given in Figure 5.1.

Since no incident force information is needed, the preconstraints can be evaluated and applied by the controller at the beginning of the time cycle before any inter-object collisions are resolved.

Preconstraint Type	Force Dependent on
spring	object position
damped spring	object position, velocity
boundary penalty	object position
air friction	translational & angular velocity

Figure 5.1 - Table of Preconstraints

5.5 Postconstraints

In contrast to the preconstraints, *postconstraints react to or oppose forces incident on an object from other external sources.* These constraints must be evaluated after collisions between objects have been considered. Other physical objects or constraints may have exerted forces which should be opposed by a postconstraint. If left unopposed, the external force will cause a constrained object to move in a way inconsistent with its constraint.

The first example developed in the PM system was the rigid bar constraint mentioned previously in Chapter Four. A table of example postconstraints is shown in Figure 5.2.

Postconstraint Type	Force Dependent on
rigid bar	velocity, time slice, forces
normal force	previous forces
friction	previous forces, relative velocity
pin or anchor post	previous forces

Figure 5.2 - Table of Postconstraints

5.6 Connecting Constraints

The approaches described above can be demonstrated effectively using the PM system, but has the drawback that the constraint objects must, in some cases, perform a considerable amount of geometry to decide where their connection points are before beginning to actually apply the constraint rule embedded in the instance methods. Implemented in the way described , a constraint would have to query each object for its position, size, and orientation. Then the constraint instance would need to perform vector math to find the location of the endpoints before calculating its force. In order to allow constraints to be more "pure" and isolated from physical details, *connectors* were developed. Proposed in [Witkin 90], this idea has been used by the PM system as well.

5.7 Connection Points

The preconstraint class *PMAnchoredSpringWithOffset* can directly perform all necessary math to connect from a point in space to an object where the object connection point is not directly at the object's center of mass. This requires a complex *apply* method and still cannot handle the case between two movable objects.

To abstract the geometric details from the constraints, the class *PMPoint* and its subclasses including *PMConnectionPoint* were developed. Instead of attaching itself directly to an object, a constraint will attach itself to instances of one of these classes. The points are examples of *connectors* which isolate the constraints from geometry and facilitate the development of a more general library of constraints.

Instances of *PMPoint* can be given a position and velocity of their own in the simulation; they will continue to move according to their initialization during the simulation and will not be involved in any collisions with physical objects. Instances of *PMConnectionPoint* can be attached to an object with any offset vector from the object's center (see Figure 5.3). During the simulation, the points automatically update their positions according to the object's motion. They are analogous to "markers" or "hooks" on a certain place on their object.

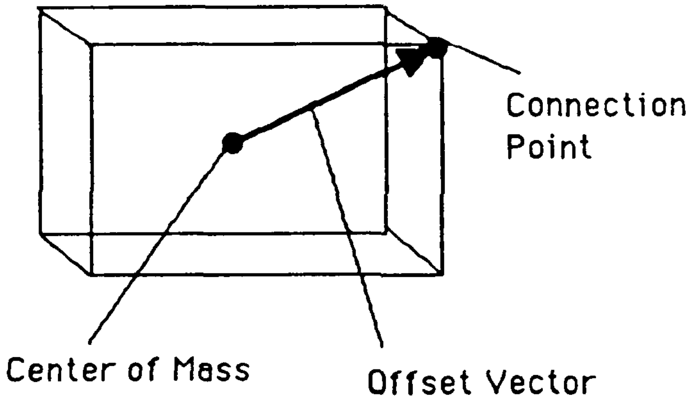


Figure 5.3 - A ConnectionPoint on an object

It follows from this that constraints can be developed which connect only to points. The preconstraint *PMSpringBetweenPoints* is an example of this type of constraint. Using positions derived by querying the points, it has a much simpler *apply* method which involves only the calculation of the constraint force.

5.8 Activating Constraints

The issues of time complexity and division of labor arise when considering if the controller should always apply the constraints at every time slice, or if the constraints should know when to apply themselves. Consider a normal force constraint existing between two objects. This normal will only apply force when the objects are in contact. If the controller calls the normal force constraint every time slice, the normal force object must know, from information requested from its constrained objects, that it should not actually exert forces unless the constrained objects are in contact. This requires an additional collision test by the constraint every invocation which is time consuming.

The alternative method involves adapting the controller to keep a list of all collisions that have ocurred this time frame and choosing one of the two following approaches:

1. Activating only the constraints which are involved with the colliding objects. Constraint objects must be cross-referenced with the physical objects they constrain for the controller to know which constraints are active. This requires more extensive intialization of the controller when the scenario is constructed, but is time effective during actual simulation (where efficiency is really needed).

2. Having the controller keep only a list of which physical objects have collided during this time cycle. It can pass this list to all constraints that need to decide whether they will apply themselves or not. It is then the constraint object's responsibility to make the decision. This is more time consuming than approach #1, but still does not require the redundant collision testing needed if the controller applies every constraint every time slice.

Connectors, as described in section 5.6, may offer a solution to this problem. Further research is needed here before a decision can be made on the best approach.

Section Seven: Extending the PM System

During the earlier chapters of this report over specific portions of the PM system, some areas for improvement have been noted. In this chapter, possible extensions and applications for the PM system will be discussed.

7.1 Multi-Object Collisions

In Chapter Four, it was mentioned that the PM system could not accurately handle multi-object collisions. The PM system classifies as a *propagation system* according to [Baraff 89], meaning that simultaneous collisions are simulated by a sequence of two-objects-at-a-time collisions (refer to the discussion of the momentum toy in Chapter Four). This approach can sometimes result in incorrect final results.

7.1.1 Simple Extensions for Multi-Object Collisions

Two extensions could be added to the basic PM system which will make it more suitable for ha~~n~~dling multiple-object collisions. Each of these could be implemented directly as subclasses to the existing *PMController* and physical object classes. Both approaches will use Figure 7.1 as an example of how they are applied.

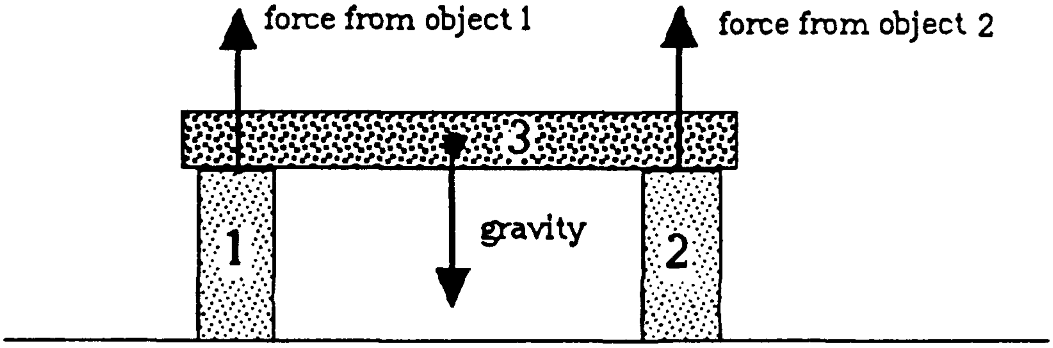


Figure 7.1 - Multiple-object Collision

The approaches are summarized as follows:

1) Extra time sub-cycle: The existing two sub-cycle approach could be extended to include a third sub-cycle which would occur previous to the other two already described. In the new sub-cycle, dubbed the *collision detect sub-cycle*, all the collisions between objects would be discovered. Then, during the force exchange sub-cycle, each object would know how many objects it is interacting with and can adjust or scale the forces it exerts appropriately. For the case in Figure 7.1, Object3 would find that it is involved in simultaneous collisions along the same axis with Object1 and Object2. Object3 would then divide the magnitude of the force by the number of objects and exert half as much force on each supporting object.

2) Objects remember collisions: Each physical object could keep a record of objects it has resolved a collision with during the current time slice. If a collision which occurs later would change the result already determined for a previous collision, the object could *fix* the first collision. It would calculate the needed change and send an additional (cancelling and fixing) force to the object involved in the first (incorrectly calculated collision). For the Figure 7.1 case here, a normal collision is resolved, assume Object1- Object3, then when the Object2 - Object3 collision occurs, Object3 will also send a cancellation force to Object1. The same final result is achieved.

Problems exist in both of these cases since the objects will need to decide which collisions affect each other and which collisions don't. For example, for Figure 7.2, the two collisions on Object1 can both be handled correctly if treated separately. Some geometrical tests must be devised to determine when to use a multiple-equation method like 1) or 2) and when to use the ordinary two-at-a-time collisions. Any such test would be subject to many special cases and may not be rigorous enough.

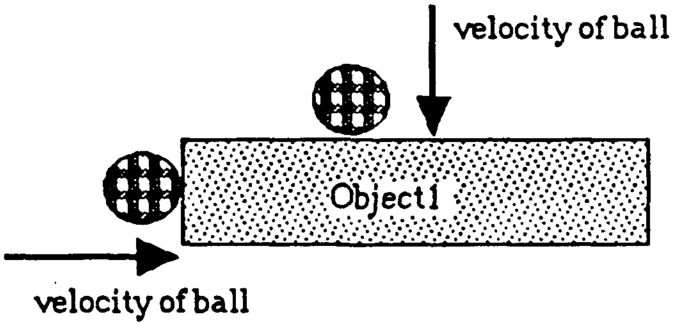


Figure 7.2 - Independent collisions

7.1.2 Including an Equation Resolver

The solutions presented in the previous section, while they may work for some collision cases, are not the most promising direction in which to extend the PM system. In [Barzel 88], constraint forces for the simulation are calculated by a simultaneous equation solving method. This approach will be more rigorous since, once the system is expressed in a system of equations, all the forces can be evaluated without testing special cases of object position and orientation (as might be necessary in the previous section's approaches).

Also, the methods in [Baraff 89] for determining normal forces for objects in constant contact are based on simultaneous equation solving. [Witkin 88] also uses a linear system solver to resolve energy and constraint equations. To include some of these features in the PM system, an equation solving system must be added.

The concept is not difficult to implement in the PM system, since subclasses of the physical object classes could be made which send their state to the equation solver instead of resolving the collision themselves. The solver could be integrated into the PM system environment as an object in Smalltalk, possibly usable by different objects in the system for different purposes.

7.2 A Look at Networked Simulation

The PM system was designed with a goal of fitting into a networked simulation involving a large number of physical objects maintained across a number of different workstations. If this type of simulation is to be accomplished, the two areas in the following sections must be considered.

7.2.1 Distributed Objects

If multiple workstations are active in a single simulation, then simulation objects, which need to communicate with each other, may exist on physically separate workstations. It is helpful in this case to provide one additional object type, a *shadow object* which will receive a message and echo it to its corresponding real object on a different workstation. See Figure 7.3 for a pictorial view of this. With this scheme, each object thinks it is communicating with a local object, and the shadow objects are serving as gateways between the workstations.

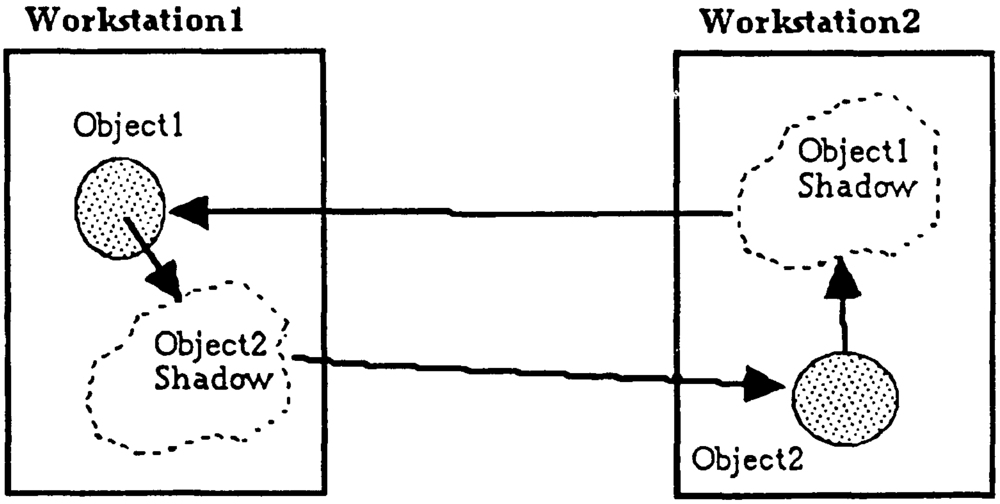


Figure 7.3 - Shadow objects

This type of simulation may affect the way which physical objects need to interact: what type of messages they should send, should acknowledgements be received, etc.

The monitor objects mentioned in Chapter 2 can be extended for use in this scenario. A monitor object could be tasked by the controller to update the state variables of any shadow objects on other workstations when necessary. When extended, monitors may be a powerful method of interfacing the PM system with other software systems in a large-scale simulation.

7.2.2 Inter-Object Messages

If objects exist on separate workstations, any messages between them must travel across the network between the workstations. This is a disadvantage since both the transfer speed and the bandwidth of today's networks are relatively low. Low bandwidth means it is easy to saturate or flood the network with messages, causing further delays.

The Smalltalk methods in the PM system create many temporary objects (mostly vectors) and perform a substantial amount of inter-object communication. *A design decision to limit the number of messages between physical objects was made.* This should ease the extension of the PM system to a networked simulation. For example, consider the following Smalltalk commands. This code assumes that physical objects have a *pos* method which returns a vector type, and when a vector type receives an *x* or *y* message, it returns the corresponding component.

```
| distSquared relativePos |
distSquared = (((aPMObject pos x)-(self pos x))*((aPMObject pos x)-(self pos x))) +
        (((aPMObject pos y)-(self pos y))*((aPMObject pos y)-(self pos y))).
relativePos = (aPMObject pos) - (self pos).
```

If *aPMObject* represents another physical object, then **five inter-object messages are needed** to execute just these two lines of code. If, however, the same function is written this way:

```
|distSquared relativePos theirPos |
theirPos = aPMObject pos.
distSquared = ((theirPos x - self pos x)*(theirPos x - self pos x)) +
        ((theirPos y - self pos y)*(theirPos y - self pos y)).
relativePos = theirPos - self pos.
```

Only one inter-object message is needed to accomplish the same calculations with this approach.

Conclusion

The PM system can be considered an implementation of the Penalty method of physical modeling. The strength (ease of modeling) and weakness (small time steps) exhibited by the PM system are consistent with the penalty method results described in the previous work reviewed (see Chapter One).

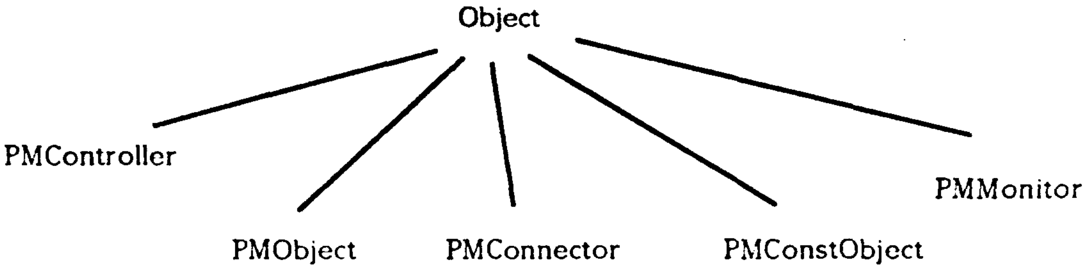
The PM system shares the approach taken by [Witkin 88] in developing a library of useful constraints. These constraints serve as tools with which to model different physical systems. The project proved experimentally that the way a physical system is modeled (how it is assembled from objects and constraints) was as important as the method used to carry out the simulation.

The object oriented design approach to physical modeling, used only by the PM system, greatly decreased the development time necessary and provided early results. When first developed, the parent classes in the PM system were instantiated in simulations. This provided necessary feedback early in the design cycle and allowed the system to evolve quickly. Subclasses inherited features from the parent classes and included additional features of their own.

The future for the PM system looks good. Because of its object oriented nature, it is easily expandable. Some improvements in collision detection and collision resolution mentioned in the paper have already begun. These can be implemented as new subclasses of physical objects – extending the concept of the PM system.

Appendix A: Smalltalk Class Descriptions

The class hierarchy shown below represents the top-level Smalltalk classes currently existing in the PM System. Simulations are composed of instances of these classes (or their subclasses) interacting. Each of the classes are described later.



A.1 PMController

An instance of this class controls a physically modeled scenario consisting of any number of interacting objects. The instance maintains a list of the objects, and uses this list to look for collisions between objects. If collisions occur, the controller instructs the objects to resolve the collision (by whatever means provided in the objects). A single view of the scene, which is a parallel projection looking at the XY plane is displayed for the user.

Methods are provided to add objects, play a scenario, add gravity, advance the time clock, and find collisions between objects. The subclasses of PMController, shown below, have additional features outlined in the following sections.

A.2 PMObject

All physical (non-constraint) objects in the simulation are instances of this class or one of its subclasses. In one sense, the class *PMObject* can be considered to be a deferred class since many common features are inherited from *PMObject* by its subclasses [Meyer 88]. However, on the other hand, instances of class *PMObject* are given enough ability to behave in simple simulations. This idea is contrary to the rule that instances cannot be made of a deferred class.

A.3 PMConnector

This defered class provides a way to isolate the geometric calculations from the constraint instances. Constraint objects are attached to instances of *PMConnector* which are, in turn, attached to a physical object. The connector provides a conceptual isolation between objects and constraints working on them.

A.4 PMConstObject

This is a deferred class which serves as the parent class for all constraint objects. No instances can be made of this class. A class hierarchy of constraint subclasses which inherit from *PMConstObject* includes several types of both preconstraints and postconstraints including springs, normal forces, air friction, and boundary conditions.

A.5 PMMonitor

The *PMMonitor* class currently has two subclasses, both designed to interpret information about the simulation from the object lists passed to them by the *PMController*. The class *PMMonitor* itself is an abstract class containing only methods which are common to its subclasses. The first subclass, *PMScenarioRecorder*, creates a textual output file which allows the calculated animation to be played back at higher speed using an animation package developed by the author [Lisle 90]. The second subclass, *PMScenarioWindow*, is part of the interface of the PM system and draws several orthogonal projections of the animated scene during calculation.

Bibliography

[Baraff 89]: David Baraff, "Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies", SIGGRAPH'89 Proceedings, ACM SIGGRAPH, 1989, pp. 223-231.

[Barr 89]: Alan H. Barr, "Teleological Modeling", SIGGRAPH'89 Physical Modeling Course Notes, ACM SIGGRAPH, 1989, pp. B1-B7.

[Barzel 88]: R. Barzel, A. Barr, "A Modeling System Based on Dynamic Constraints", SIGGRAPH'88 Proceedings, 1988.

[Barzel 89]: Ronen Barzel, Alan Barr, "Controlling Rigid Bodies with Dynamic Constraints", SIGGRAPH'89 Physical Modeling Course Notes, ACM SIGGRAPH, 1989, pp. D1-D17.

[Beer 88]: Ferdinand P. Beer, E. Russel Johnston, Jr., Vector Mechanics for Engineers: Dynamics, McGraw-Hill, New York, 1988, pp.614-635, 785-788, 891-908, 974-975.

[Blinn 87]: James Blinn, "*The Mechanical Universe:* An Integrated View of a Large Scale Animation Project", SIGGRAPH'87 Course Notes, ACM SIGGRAPH, 1987, pp.61-81.

[Bueche, 88]: F. Bueche, Principles oi Physics, Mc-Graw Hill, New York, 1988, pp. 137-146,160-162

[Ezzell 90]: Ben Ezzell, "Writing Reusable Objects", Computer Language, June 1990, pp.34-43.

[Goldstein 50]: Herbert Goldstein, Classical Mechanics, Addison-Wesley, Reading, MS, pp.143-163.

[Greenwood, 88]: Donald Greenwood, Principles of Dynamics, Prentice-Hall, Englewood Cliffs, New Jersey, 1988, pp.100-101,145-146,157-160, 389-392.

[Isaacs 87]: Paul M. Isaacs, Michael F. Cohen, "Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions and Inverse Dynamics", Computer Graphics, Vol. 21, No. 4, July 1987.

[Lisle 90]: Curtis Lisle, "Design Document for ANIM: A Simple Animation Controller", VSL Memo 90.10, The Institute for Simultation and Training, University of Central Florida, Orlando, FL, July 1990.

[MacMillan 36]: William D. MacMillan, Dynamics of Rigid Bodies, Dover Publication, Inc. New York, 1936.

[Moore 88]: Matthew Moore and Jane Wilhelms, "Collision Detection and Response for Computer Animation", Computer Graphics, August 1988, pp. 289-298.

[Moshell 90 #1]: J. Michael Moshell, "Work Plan for Dynamic Terrain Project: *The Virtual Reality Testbed* ", VSL Memo 90.5, The Institute for Simultation and Training, University of Central Florida, Orlando, FL, March 1990.

[Moshell 90 #2]: J. Michael Moshell, Charles Hughes, Brian Goldiez, Brian Blau, Xin Li, "Dynamic Terrain in Networked Visual Simulators", VSL Document 89.17, The Institute for Simultation and Training, University of Central Florida, Orlando, FL, January 1990.

[Meyer 88]: Bertrand Meyer, Object-Oriented Software Construction, Prentice Hall, New York, 1988, pp. 230,52-60.

[Ohanian 89]: Hans C. Ohanian, Physics: 2nd Edition, W.W. Norton & Co., New York, 1989, pp. 74-198, 335-345.

[Platt 89]: John Platt, "Constraint Methods for Physical Systems",SIGGRAPH'89 Physical Modeling Course Notes, ACM SIGGRAPH, 1989, pp. C1-C14. (this is an excerpt from Platt's Cal-Tech Computer Science PhD Thesis entitled *Constraint Methods for Neural Networks and Computer Graphics*).

[Rabenstein 82]: Albert L. Rabenstein, Elementary Differential Equations with Linear Algebra (3rd Edition), Academic Press (division of HBJ), New York, 1982, pp. 419-424.

[Rogers 85]: David F. Rogers, Procedural Elements for Computer Graphics, McGraw-Hill Book Company, New York, 1985.

[Serway 86]: Raymond Serway, Physics for Scientists and Engineers, 2nd Edition, Saunders College Publishing, Philadelphia, 1986. pp. 200-227.

[Witkin 90]: Andrew Witkin, Michael Gleicher, William Welch "Interactive Dynamics", ACM Computer Graphics, 1990.

[Witkin 89 #1]: Andrew Witkin, Michael Kass, "Spacetime Constraints", SIGGRAPH'89 Physical Modeling Course Notes, ACM SIGGRAPH, 1989, pp. E1-E10.

[Witkin 89 #2]: Andrew Witkin, Kurt Fleischer, Alan Barr, "Energy Constraints on Parameterized Models", SIGGRAPH'89 Physical Modeling Course Notes, ACM SIGGRAPH, 1989, pp. F1-F8.

Appendix H:

Realtime Hydrology

Dynamic Terrain Project
1990-91

Fluid Dynamics Methodologies for Computer Graphics

by
Charles E. Campbell

Graduate Committee:
Dr. J. Michael Moshell (Chairman)
Dr. Charles E. Hughes
Dr. Ali Orooji

Submitted in partial fulfillment of the requirements
for the Master of Science degree in Computer Science in the
Graduate Studies Program of the College of Arts and Sciences
University of Central Florida
Orlando Florida

Fall 1991

Abstract

This paper describes the methods used to provide a realtime simulation of fluid flow to the IST/UCF dynamic terrain project. Several methods for fluid animation were researched, and one was chosen for use in this project.

While the fluid model was implemented as part of this project, the main focus was extending the model to accept floating bodies. This involved fluid displacement and the creation of ripples to simulate disturbances in the water caused by floating objects.

To demonstrate the usefulness of a water model to the dynamic terrain project, a simulation of an army ribbon bridge was constructed. The bridge is made up of individual sections which interact to create a realistic effect as a vehicle crosses the river.

Some thoughts on the incorporation of the water model into a distributed database management system are discussed. And finally, some insight is given into extensions and applications for the water model.

Acknowledgements

I would first and foremost like to thank Dr. J. Michael Moshell for everything he has done for me throughout my graduate studies. He has made my course work interesting and enjoyable, and the opportunities and experiences he has presented me with at IST are invaluable. I would also like to thank Dr. Charles E. Hughes and Dr. Ali Orooji for their participation in my graduate experience.

I would also like to thank Xin Li for sharing his knowledge of physical modeling, and everyone else at IST for their ideas, advise, and most of all friendship. They have taught me not to take life too seriously and that nothing is impossible. Kudos to you all!

Thanks to my family both in Orlando and back home in the Midwest for all their encouragement and support. Thanks also to John, Chris, and Jeff for their encouragement and for a lifetime of friendship.

Finally, thanks to my lovely wife, Melissa, for her patience and understanding through it all. She is the most important part of my life and I love her with all my heart.

Table of Contents

Chapter 1
Introduction

1.1 Dynamic Terrain

Existing ground combat training simulators have dealt mainly with vehicle and weapon dynamics. As advancements in hardware became available, simulations began to focus on more sophisticated topics, such as intelligent behavior, learning systems, and networked simulations. One area currently being researched for the next generation of simulators is **dynamic terrain**.

Traditional terrain databases are static and cannot be modified during a simulation. Thus, bombing a bridge has no effect on the database, and pushing over trees is impossible. Some simulators allow a limited form of altering the database, such as the ability to blow up a bridge, but the behavior is always the same and only applies to those objects which are specifically programmed to behave a certain way in a given simulation.

To increase the effectiveness of simulator training, the simulation needs to more accurately reflect real world conditions. In a traditional ground-based warfare simulation, many aspects of actual battlefield tactics are ignored because of the lack of simulator ability. Important activities such as digging anti-tank trenches and gap-crossing are neglected, decreasing the realism of the simulation.

Dynamic terrain deals with the ability to modify terrain attributes during a simulation. Modifications are not only performed by vehicle actions, such as cannon fire, but can also be due to weathering effects, such as rain and snow. Substantial research has been performed in the area of dynamic terrain at IST [Moshell 92, Hua 91, Moshell 90], although most of the work has involved changes due to vehicle actions.

This project deals with the incorporation of a realtime, computationally tractible fluid model into the IST/UCF dynamic terrain project. The goal is to lay the groundwork for increasing simulation realism through numerous applications involving fluids. By using a versatile fluid model, the work described in this document should be easily extendable to a number of fluid related topics not covered within the scope of this project. In a larger context, this probject is part of an extended effort at UCF to contribute to the state of the art in physically-based computer graphics.

The introduction of fluids raised a number of interesting computational problems for distributed simulation with multiple databases. This topic is also addressed by this project.

1.2 Background

Gaps have always provided a major obstacle for mobile ground-base armies. Traditional simulators provide means for gap crossing in the way of bridges. However, in a dynamic terrain simulation, bridges can be destroyed. Therefore, a realistic simulation should provide a means for crossing gaps without the means of static structures. Because this project involves the addition of a fluid model to simulations, gap crossing maneuvers will focus on rivers and streams, which is a very complex military operation [FM5-101].

With the absence of static structures with which to cross a body of water, alternate methods must be explored. Fording the river is the most obvious approach to a river crossing, but is not always possible. An armored vehicle launched bridge (AVLB) is another expedient method of gap crossing, but has a strict limitation on the size of the gap it is able to span (63 feet). Also, the AVLB can only be placed at locations where the slope of the river bank and the soil composition provide an adequate foundation.

This leads to the idea of floating vehicles and troops across the river. Heavy rafts are quick to assemble and can be launched from multiple sites. However, rafting is not suitable for crossing a large volume of traffic in a short amount of time.

To rapidly cross large volumes of traffic, a bridge is needed. But bridges are very time consuming to build and require a great deal of manpower and materials. Suppose, however, some of the heavy rafts could be linked together to form a floating bridge. This is the idea behind the **ribbon bridge**.

The ribbon bridge is the primary assault bridge because it is quick to assemble [FM90-13]. It is composed of modular sections made of aluminum alloy and can be constructed under all conditions of visibility [FM5-101], and provides a solution to the problem of getting an army across a body of water quickly.

1.3 Project Description

This project involves the incorporation of a fluid model into the dynamic terrain project. The fluid model must be versatile enough to handle a variety of applications in order to create a realistic simulation.

To demonstrate the benefits of a fluid model to a simulation, this project focuses upon the simulation of a ribbon bridge, with vehicles of varying weights driving across. This involves such subtopics as water displacement, floating bodies, and constraint-based modeling.

Also, an investigation of the incorporation of the fluid model into a distributed database management system (DDBMS) is presented.

Chapter 2
The Water Model

This chapter presents a brief overview of some of the previous attempts at fluid modeling, then goes on to explain in detail how the fluid model of Kass and Miller [Kass 90] works and why it is appropriate for this project.

2.1 Overview of Fluid Simulation

The topic of fluid flow and water modeling has drawn a lot of attention in recent years. The goal is to produce a model which is faithful to the laws of hydrodynamics, yet simple enough to provide a realistic animation in real time. This is not an easy task given the complexity of the equations governing even simple fluid flow.

One of the first attempts at water wave animation was created by Nelson Max [Max 81] in 1981. He represents the ocean as a parametric surface using conflicting sine waves in his film "Carla's Island". While the overall effect is pleasing, the model has little to do with hydrodynamics.

In 1986, Fournier and Reeves [Fournier 86] presented a method which more realistically follows the laws of hydrodynamics. Their model uses the concept of *orbitals*. Water particles travel in a circular motion, which becomes more elliptical as the waves move toward the shore and the water becomes more shallow. This produces breaking waves. Particle systems have also been used to add foam and spray to the system [Sorensen 91]. This model produces beautiful still pictures, yet the amount of computation for each iteration makes a real time simulation beyond the reach of current computer hardware.

Also in 1986, Darwyn Peachy [Peachy 86] introduced an ocean model to produce realistic images of waves on a beach. He uses a *phase function* to model wave refraction and the change of the speed and wavelength in shallow water in conjunction with a *wave profile* which is a collection of vertical displacements of the ocean's surface from the rest position and changes due to the shape of the wave and the depth of the water. Again, this model produces stunning images, but is not practical at this time for a real time simulation.

Other methods of modeling fluid flow include finite elements [Steven 78], penalty solution [De Bremaecker 87], and particles simulation [Choquin 89]. Yet none of these methods are appropriate for the task at hand.

Finally, in 1990 Kass and Miller [Kass 90] presented a method for animating fluids based upon an approximation to shallow water equations. Their method handles wave refractions, wave reflections, and the net transport of water. Thus it provides the versatility needed for incorporation into the dynamic terrain model. Most importantly, the simplified equations can be solved quickly enough for real time animation.

The remainder of this chapter describes our implementation of Kass and Miller's model. Our extensions to the water model are introduced in Chapter 3.

2.2 The Water Model

The model introduced by Kass and Miller treats the water surface as a height field over a uniform grid. Each grid point contains a known volume of water. At each iteration, gravity attempts to pull down the higher columns. The direction and velocity of each column of water is known, so the volume of the water can be conserved by transporting it to neighboring columns at the proper speed and heading.

This is all accomplished by solving a set of partial differential equations derived from a simplified set of shallow water equations. A discrete representation of these equations is constructed using the finite-difference technique. The resulting differential equation is further simplified to make it linear. The system of linear equations to be solved is tridiagonal, so the amount of computation per iteration is proportional to the number of grid points. Each of these steps are explained more fully below.

Obviously the many simplifications introduced reduce the accuracy of the simulation. However, these simplifications must be performed in order to produce a sufficient frame rate to achieve a real time animation. The resulting animatio.. is realistic to the eye as long as the limitations of the model are not exceeded.

2.3 Shallow Water Equations

The shallow water equations used in the water model are not new, and can often be found in texts on water waves [Stoker 57, Whitham 74]. They are derived from the full Navier-Stokes equations for fluid flow. Three approximations bring about their simplified form. First, the water surface is treated as a height field. This approximation makes it impossible for the model to produce breaking waves because they would require multiple height values. The second approximation assumes the vertical velocity of a particle of water can be ignored. This assumption causes the accuracy of the water model to degrade if the waves become very steep. Finally, they treat the horizontal velocity of a column of water as approximately constant. Conditions which demand the horizontal velocity to drastically increase or decrease will again decrease the accuracy of the model. These assumptions place some obvious limitations on the accuracy of the water model, but for most conditions, they will have little effect on the visual realism of the simulation.

To illustrate how the model works, the method is described in two dimensions (see fig.1). For a sample point x, b_x is defined as the altitude of the terrain surface and h_x the altitude of the water surface. Then, the water depth can be stated as $d_x = h_x - b_x$. Finally, u_x is the horizontal flow between column d_x and column d_{x+1}.

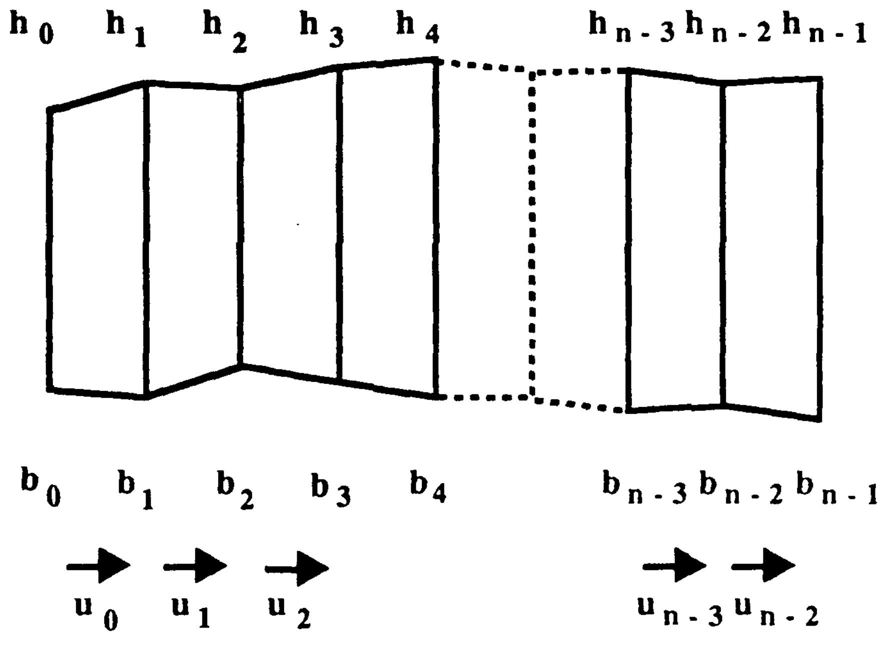

h_0 h_1 h_2 h_3 h_4 h_{n-3} h_{n-2} h_{n-1}
b_0 b_1 b_2 b_3 b_4 b_{n-3} b_{n-2} b_{n-1}
u_0 u_1 u_2 u_{n-3} u_{n-2}

fig. 1

The shallow water equations themselves can be written:

$$\frac{\delta u}{\delta t} + g\frac{\delta h}{\delta x} = 0 \qquad \text{(eq. 1)}$$

$$\frac{\delta h}{\delta t} + d\frac{\delta u}{\delta x} = 0 \qquad \text{(eq. 2)}$$

where g is the gravitational constant equal to 9.8 m/s^2.
The first equation states that the velocity of a column of water is determined by gravity acting upon it. The second states that the height of the water surface depends upon the depth of the water and the speed and direction it is moving.

Differentiating eq. 1 with respect to x, eq. 2 with respect to t, and substituting yields

$$\frac{\delta^2 h}{\delta t^2} = gd\frac{\delta^2 h}{\delta x^2}$$

(eq. 3)

which is the one-dimensional wave equation with wave velocity $\sqrt{gd}$ [Stoker 57].

2.4 The Finite-Difference Technique

The problem now is to solve eq. 3. One widely used method of dealing with partial differential equations is the finite-difference technique. This technique is base on the theorem which states that any curve can be divided up into nearly straight segments with sufficiently small divisions [Reece 86].

To start, consider a first derivative, where samples in the x-direction are uniform:

$$\frac{\delta h}{\delta x} = \frac{\text{increase in h}}{\text{increase in x}} = \frac{h_i - h_{i-1}}{\Delta x}$$

(eq. 4)

This approximation can be applied to eqs. 1 and 2 to derive

$$\frac{\delta u_i}{\delta t} = -g\frac{(h_{i+1} - h_i)}{\Delta x}$$

(eq. 5)

$$\frac{\delta h_i}{\delta t} = \frac{(d_i + d_{i+1})}{2\Delta x} u_i + \frac{(d_{i-1} + d_i)}{2\Delta x} u_{i-1}$$

(eq. 6)

with flow rate u_i being calculated between columns h_i and h_{i+1} as previously stated, and d_i the depth of the fluid in a column.

Putting these equations together produces the finite difference approximation to eq. 3

$$\frac{\delta^2 h_i}{\delta t^2} = -\left(\frac{d_{i-1} + d_i}{2\Delta x}\right)\frac{\delta u_{i-1}}{\delta t} + \left(\frac{d_i + d_{i+1}}{2\Delta x}\right)\frac{\delta u_i}{\delta t}$$

$$= -g\left(\frac{d_{i-1} + d_i}{2(\Delta x)^2}\right)(h_i - h_{i-1})$$

$$+ g\left(\frac{d_i + d_{i+1}}{2(\Delta x)^2}\right)(h_{i+1} - h_i) \qquad \text{(eq. 7)}$$

2.5 Integration

The next step in the process is to solve the differential equations resulting from applying the finite-difference technique to the original partial-differential equations. For this, a first-order implicit method is used. Using h' and h'' to denote differentiation with respect to time, and n to denote the nth iteration, the first order implicit equations are:

$$\frac{h(n) - h(n-1)}{\Delta t} = h'(n) \qquad \text{(eq. 8)}$$

$$\frac{h'(n) - h'(n-1)}{\Delta t} = h''(n) \qquad \text{(eq. 9)}$$

Calculating h' and h'' at the end of the iteration (time n) instead of the beginning of the iteration (time n-1) makes the iteration implicit and stable [Kass 90].

Solving for h(n) in eqs. 8 and 9 gives:

$$h(n) = 2h(n-1) - h(n-2) + (\Delta t)^2 h''(n) \qquad \text{(eq. 10)}$$

Substituting eq. 7 into eq. 10 yields

$$h_i(n) = 2h_i(n-1) - h_i(n-2) \qquad \text{(11a)}$$

$$- g(\Delta t)^2 \left(\frac{d_{i-1} + d_i}{2(\Delta x)^2} \right) (h_i(n) - h_{i-1}(n)) \qquad \text{(11b)}$$

$$+ g(\Delta t)^2 \left(\frac{d_i + d_{i+1}}{2(\Delta x)^2} \right) (h_{i+1}(n) - h_i(n)) \qquad \text{(11c)}$$

$$\text{(eq. 11)}$$

which is still non-linear because the column height, d, depends on the surface height, h.

In Equation 11, the terms can be physically interpreted as follows [Moshell 91a]:

> The overall equation is intended to be integrable so as to compute the new height of a water column, and thus must measure the net effect of water flowing in and out of the column. Term 11a represents the history of the column before the current simulation step; its precise form is a relic of two-stage Euler integration.
>
> Two factors influence the rate of flow into a container: the size of the aperture and the pressure differential driving the flow. We can regard the open sides of a water column as the aperture, and the height difference between this and adjacent columns as indicators of pressure difference.
>
> Term 11b concerns flow between columns i and i-1. Term 11c, for flow between columns i and i+1, is similar. The expression $((d_{i-1} + d_i)/2(\Delta x)^2)$ in term 11a measures the effective height of a column of water (bottom to surface), divided by its horizontal area. The effect of this term is to reflect the fact that

a taller column would have a larger external surface area, and thus would offer less resistance to in- and out-flow; whereas a "thicker" column (larger horizontal area) would offer relatively more resistance, since water within the column is obstructed by water on the outside.

The expression $(h_i(n) - h_{i-1}(n))$ in term 11b measures the difference in height of this and adjacent columns, and thus the driving pressure differential.

In a continuous integral, the aperture would change continually along with the height of the water surface; thus, the net inflow computation during a time interval would yield a nonlinear time history (figure 2) for the surface height. However this quadratic situation makes the simultaneous solution of large numbers of coupled cells in a discrete-time simulation too expensive.

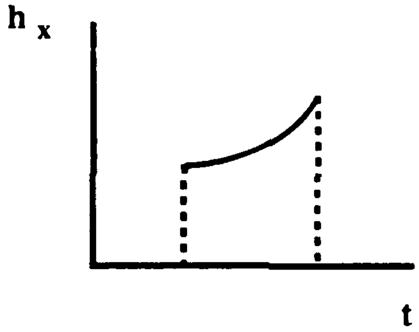


Figure 2: Time history of surface height for continuously varying aperture

A slight simplification is to fix the size of the "aperture" during one time slice. Thus, the flow into or out of a cell is still proportional to the cell's recent aperture, but the flow rate is constant during the time interval, and a picture of the time history of the surface for the discrete-time simulation would look like figure 3.

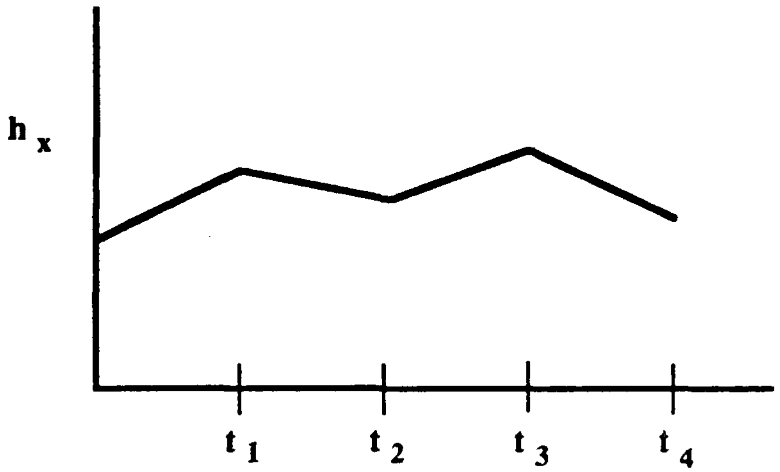


Figure 3: Time history of surface height for piecewise static aperture

It is important to note that the hydrology model remains nonlinear in the overall sense: flow into cells is proportional to both aperture and pressure difference. However we have linearized the simultaneous equations which must be solved to establish the surface profile across one time step.

Thus, to linearize the equations, d is treated as a constant during one step of the iteration. Now h(n) can be calculated from previously calculated values in a *tridiagonal* system of equations.

A tridiagonal system of linear equations has nonzero elements only on the diagonal plus or minus one column [Press 86]. Because of its nature, the solution (using forward- and backsubstitution) takes only O(n) time. The tridiagonal matrix is derived rearranging eq. 11 to the following form:

$$\left(1 + g(\Delta t)^2\left(\frac{d_{i-1} + 2d_i + d_{i+1}}{2(\Delta x)^2}\right)\right) h_i(n)$$

$$- g(\Delta t)^2 \left(\frac{d_{i-1} + d_i}{2(\Delta x)^2}\right) h_{i-1}(n)$$

$$- g(\Delta t)^2 \left(\frac{d_i + d_{i+1}}{2(\Delta x)^2}\right) h_{i+1}(n)$$

$$= 2h_i(n-1) - h_i(n-2)$$

(eq. 12)

The left hand side of the equation is used to form the tridiagonal matrix, with the first term of the equation being the diagonal. Thus, the left hand side of the equation can be reduced, yielding

$$A h_i(n) = 2h_i(n-1) - h_i(n-2)$$ (eq. 13)

where the matrix A is constructed as follows (from eq. 12):

$$e_0 = 1 + g(\Delta t)^2\left(\frac{d_0 + d_1}{2(\Delta x)^2}\right)$$

$$e_i = 1 + g(\Delta t)^2\left(\frac{d_{i-1} + 2d_i + d_{i+1}}{2(\Delta x)^2}\right) \quad (0 < i < n\text{-}1)$$

$$e_{n-1} = 1 + g(\Delta t)^2\left(\frac{d_{n-2} + d_{n-1}}{2(\Delta x)^2}\right)$$

$$f_i = -\ g(\Delta t)^2\left(\frac{d_i + d_{i+1}}{2(\Delta x)^2}\right)$$

$$A = \begin{bmatrix} e_0 & f_0 & & & & & \\ f_0 & e_1 & f_1 & & & & \\ & f_1 & e_2 & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & e_{n-3} & f_{n-3} & \\ & & & & f_{n-3} & e_{n-2} & f_{n-2} \\ & & & & & f_{n-2} & e_{n-1} \end{bmatrix}$$

(eq. 14)

One final modification to the equations can be performed. By altering eq. 13 to the form

$$A\,h_i(n) = h_i(n\text{-}1) + (1\text{-}\tau)(h_i(n\text{-}1) - h_i(n\text{-}2)) \qquad \text{(eq. 15)}$$

where $0 \le \tau \le 1$, a damping force on the waves is added to simulate viscosity. If $\tau = 0$, eq. 15 is equivalent to eq. 13.

2.6 Vo!ume Conservation

The simplifications applied to the original shallow water equations to derive eq. 13 have one major drawback - the volume of water is no longer precisely conserved. An iteration may compute the height of the surface of the water to be less than that of the terrain at a particular location. This means that an excess of water will be created elsewhere. This requires the negative water heights be set to the terrain height (depth = 0) and the new volume of water adjusted to equal the old volume. This is done by reducing the new volume uniformly over the samples which contain water.

2.7 Sources and Sinks

At the start of the simulation, the surface height of the water is initialized to the surface height of the terrain. If the water height is below the height of the terrain, adding water to the terrain will not have the desired effect.

The addition and subtraction of water to the model is a relatively simple process. All that is required is a modification of the water surface height at the desired locations. Both the current and previous water height fields need to be adjusted for volume conservation.

The sources and sinks should be simulated after the drawing routine but before the next water height field is calculated. If not, the sources may appear to be solid "mounds" of water, while the sinks may appear as holes in the water surface. Also, special attention must be taken when creating sinks so that no more water is taken away from an area than exists. Volume conservation will account for the negative heights created, but the difference will be distributed over the entire volume, which does not produce the desired effect.

2.8 Algorithm

With all the pieces in place, the algorithm can finally be given. It is surprisingly straightforward and concise.

```
Set waterSurface(0) and waterSurface(1) equal to
    terrainSurface
For j = 2 to n by 1    (n is the number of iterations)
  Adjust waterSurface(j-1) and waterSurface(j-2) to
    reflect any additions or subtractions of water
  Compute the waterDepth using waterSurface(j-1) and
    terrainSurface
  Using equation 13, calculate waterSurface(j) from
    waterSurface(j-1) and waterSurface(j-2)
  Using waterSurface(j-1), adjust waterSurface(j) to
    conserve volume as discussed in section 2.6
```

2.9 Three Dimensions

The move to three dimensions is treated as an extension of the two dimensional case. The iteration is simply divided into two sub-iterations: one for the x-direction and one for the y-direction. Each iteration, the height field is computed for the rows using the two dimensional method. The result is then used to compute the height field using the columns. Some potential visual defects can arise, but they are slight, and can be justified by the speed of the algorithm.

2.10 Increasing Performance

As previously stated, this fluid model produces realtime simulations superior to other existing methods. The equations have been simplified and put into a form which is computationally efficient. Still, the vast amount of computation required each iteration is very demanding on the system, especially if the number of grid points is large.

In an attempt to increase the frame rate, the display routine and the fluid model have been separated to run in parallel on different processors, using a Silicon Graphics 240 GTX (Power Series) workstation. The single processor frame rate is just under 3 frames/second. With two processors, the frame rate has increased to just over 5 frames/second. This makes a significant difference on the overall visual effect. The terrain used in the performance test consisted of a 75x75 grid of elevation posts. Since the gravitational constant for the model is 9.8 m/s^2, the distance scale between two terrain posts is one meter.

The method of display imposes a lower bound on the maximum frame rate that can be obtained. The entire display routine must be performed on the processor which opens the display window. The display method used for this project (flat shading for both the terrain and the water polygons) takes only slightly less time than the computations for the fluid model. For these reasons, no further consideration was given to the use of additional processors.

Chapter 3
Floating Objects

To simulate objects floating in a fluid, two important issues must be addressed. The first is orientation. As waves sweep across the fluid surface, an object sitting in the fluid will be rocked back and forth, side to side, and up and down. For each iteration it is important to orient the object in the appropriate manner based upon the fluid surface so that the object appears to float. The second task is buoyancy. For an object to truly appear to float, it should sink into the fluid a distance proportional to its weight. Also, the object should displace fluid, resulting in an increased fluid level.

This chapter addresses these two issues as they pertain to rectangular objects. More specifically, a raft on a body of water will be used as an example, although the methods used will work for any rectangular object. The solutions provided are designed to be compatible with the fluid model chosen.

3.1 Orientation

The orientation of a raft on a body of water is governed by the water's surface. By sampling height values at various points along the raft, a surface normal to the water can be determined. Using this normal, the angles of rotation around the x- and y-axes can be computed. The raft can then be translated and rotated according to the values computed, creating a realistic visual effect.

The process starts by obtaining height values for points along the raft's surface. This is simply a table lookup at a known <x,y> location in the array containing the water height field. Using any three points along the surface of the raft, a surface normal to the water can be obtained using the cross product.

Obviously, arbitrarily choosing any three points and calculating the normal will not produce accurate results. A better solution is to calculate several normals using sets of points from different locations on the raft and using the average. In the interest of speed, this project computes the average of the two normals obtained from the two triangles resulting from splitting the raft's surface along the diagonal. This is still a crude approximation, yet the results are quite good.

Once the normal to the water surface is known, it is a simple exercise in trigonometry to determine the angles of rotation with which to orient the raft. Two angles, Θ and φ, need to be computed (see figure 4). This is accomplished using the cosine rule for right triangles:

$$\cos\ \Theta = \frac{\text{side adjacent}}{\text{hypotenuse}} \qquad \text{(eq. 16)}$$

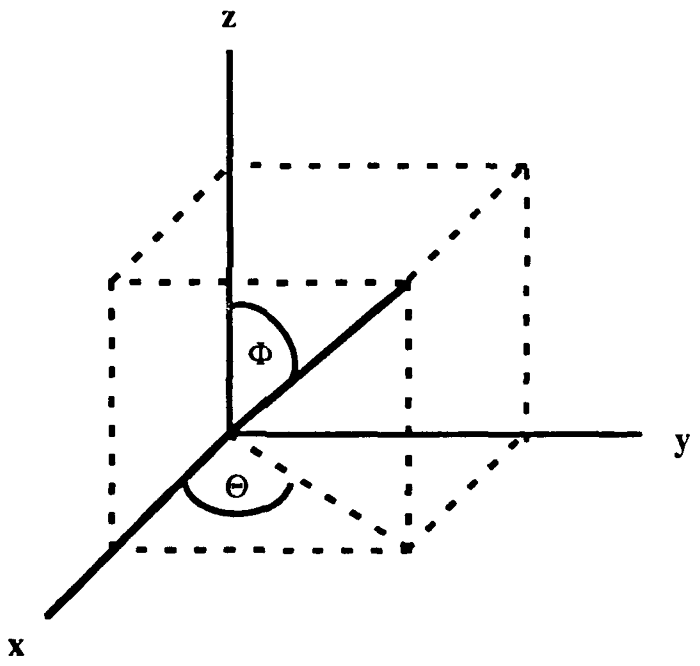


Figure 4.

For Θ, the problem is projected into the x-y plane. The length of the adjacent side is the x component of the water surface normal. The hypotenuse can be determined using the Pythagorean theorem. Thus we have

$$\Theta = \arccos\left(\frac{\text{normal}(x)}{\sqrt{(\text{normal}(x))^{2} + (\text{normal}(y))^{2}}}\right)$$

(eq. 17)

For φ, the problem is a bit simpler. The angle is between the z-axis and the water surface normal, which is a normalized vector (i.e. its length is 1). The equation is simply

$$\Theta = \arccos(\text{normal}(z)) \qquad \text{(eq. 18)}$$

The final step is to perform the rotations. To obtain the correct orientation, the sequence of rotation is as follows: rotate by Θ counterclockwise around the z-axis, rotate by φ counterclockwise around the new y-axis, and finally, rotate by Θ clockwise around the new z-axis. This will give the raft the proper orientatiun in the water.

3.2 Fluid Displacement

When an object is placed in a fluid, the total weight of the object must be supported by the buoyant force of the water. This buoyant force acts vertically upward through the center of gravity of the object when it is at rest. Archimedes' principle states that the buoyant force is equal to the weight of the fluid displaced by the object. Using this principle, the following equation is derived:

$$V\rho g = Mg \qquad \text{(eq. 19)}$$

Here V is the submerged volume of the object, ρ is the density of the fluid, g is gravity, and M is the total mass of the object.

This project deals only with rectangular floating objects. Thus the volume can be decomposed into length (l), width (w), and depth

(d) (see figure 5). Also, gravity can be factored out of the equation altogether. Thus, eq. 19 becomes

$$lwd\rho = M \qquad \text{(eq. 20)}$$

This equation is then used to calculate the volume of fluid displaced by a rectangular object as well as the depth to which the object will sink into the fluid. This of course assumes that the center of gravity of the object is at its center and that the object is at rest. While this is not always the case, this assumption is acceptable for simulation of a raft with no cargo. Extension to a more accurate simulation is presented in ch.pter 6.

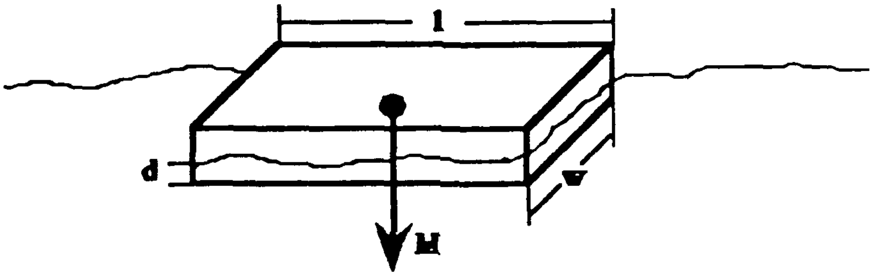


Figure 5. A raft floating in water

3.3 Other Details

This section describes some of the issues faced when creating the simulation of floating objects and how they are handled in this project.

3.3.1 Water-Object Interaction

Flowing water must interact with objects placed in it. This interaction can be very complex. Water will rebound off of the object and seek an alternate path. The water also exerts a force upon the object, which may be sufficient to give the object a nonzero velocity. Accurate interaction goes beyond the scope of this project, since the final objective is the simulation of a ribbon bridge, which is constrained to remain at the same location during the simulation. However, some discussion on the subject of velocity is given in chapter 6.

The representation of the water as a height field poses special difficulties for accurate interaction with objects floating in them. The height field representation prevents water from flowing over the top of an object, just as it prevents breaking waves. And the water height under an object must be constrained to not rise above the height of the bottom surface of the object.

A possible solution to these problems is to treat the terrain under the object as having the same height as the object's surface for the computation of the water surface. As an object moves, the section of terrain can be adjusted to remain consistent with this approach. The advantages of this approach are that the water height is constrained under the object by removing of it altogether. This approach will cause waves to rebound off of the object, and it will allow water to flow over the top of the object if the object is submerged. However, this method has many disadvantages. The water volume and height fields must constantly be adjusted to reflect the amount of water which is added and discarded by adjusting the terrain. Also, the water flow is not just disrupted by the object, but by the entire column from the object to the terrain surface. Most importantly, this method defeats the advantage of having a model based upon actual water equations and is computationally expensive. For these reasons, this method is not recommended.

Another solution considered is an obvious one: reduce the height field under the object to reflect the volume of water displaced and increase the height field of all other points which contain a volume of water to reflect the displaced volume. The advantage of this method is that it is simple and fast. Also, the water flow under the object is not obstructed as in the previous method. One disadvantage to this approach is that the water cannot flow over the top of the object. Another is that the water surrounding the object will flow toward the "gap" made by displacing the water under the object, which is unrealistic. At each iteration, the water flowing back into the "gap" will have to be removed. To counteract this problem, the height field could be constrained, but this would lead to the same problems the last method encountered.

The method chosen to model fluid displacement is to simply add the amount of fluid displaced to the total volume. The water level will rise the appropriate amount, giving a realistic visual effect. This method allows the water to flow over and under the object unobstructed. Most importantly, there are no additional computations to be made. The disadvantage is that the water flow is unaffected by the object. This is, of course, unrealistic, yet is an acceptable tradeoff considering the rebounding waves off of the object will probably be unnoticeable most of the time. The model could probably be modified to simulate rebounding waves, at additional computational cost. And, as previously stated, the effects of the water's force on the object to cause movement are explored in chapter 6. The speed and simplicity of this method and the benefits of the implementation makes this approach suitable for this project.

3.3.2 Generating Waves

Adding weight to a floating object will cause the object to sink further into the fluid. If the weight is great enough, waves will be generated outward from the object due to the displaced water. The same visual effect occurs when a great enough weight is removed from a floating object as the buoyant force pushes the object out of the fluid.

The previous section described how fluid displacement is performed. The method descrih-d is easily modified to create waves as the force on a floating object is disturbed. Instead of distributing the volume of the displaced water uniformly throughout the body of water, distribute it only along the edges of the object (see figure 6). This creates the appropriate rippling effect across the water surface, and also causes the object to bob realistically as the waves settle.

The water level rises to reflect the volume of water displaced. The effect is used for both the addition and subtraction of water resulting from an adjustment of weight (force) on an object.

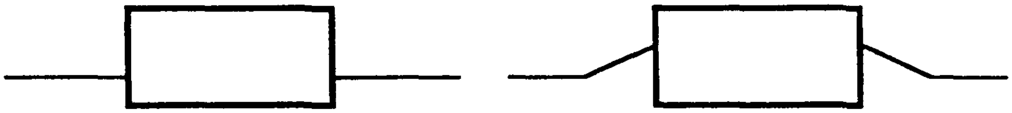


Figure 6. A raft floating in the water (a) and the instant weight is added (b).

3.3.3 Time Slicing

The fluid model calculates the flow of a fluid over time. The time is not continuous, but is divided into small slices. Thus, the state of the water surface is calculated at specific instances in time. All that occurs between these instants in time will be lost.

An actual flowing fluid can be thought of as using an infinitely small time slice. The accuracy of time slicing system is increased as the time slice is decreased. However, as the time slice decreases, the amount of computation required maintain the speed (frame rate) of the animation increases.

When dealing with the fluid model, too large a time slice means the waves will appear to dampen out too quickly and the simulation will not look realistic. With too small a time slice, the waves will not appear to move quickly enough, giving the visual effect of being "chunky" instead of fluid. For these reasons, it is important to determine a time slice which will produce a realistic looking simulation for the type of situation being modeled.

Chapter 4
The Ribbon Bridge Simulation

As stated in section 1.2, the ribbon bridge is an important part of ground-based military maneuvers. It is composed of modular components, and has the capability of transporting large volumes of traffic across a river in a short period of time.

The simulation of a ribbon bridge has several parts: the river to be crossed, the vehicles which will use the bridge to cross the river, and the bridge itself. This chapter discusses the various components of the ribbon bridge simulation, how they are modeled, and how they interact.

4.1 Modeling the River

The dynamics of the river do not play an important role in the simulation of the ribbon bridge. As stated in the last chapter, floating objects are only aware of a buoyant force acting upon them. Thus, there is no horizontal force on the bridge from the water.

For this reason, the river does not have to be modeled as flowing water. It could be thought of as a pond. Still, it is desirable to have a realistic looking simulation. And, since this approach does not require much additional computation, the river is modeled with flowing water.

This is accomplished by simply creating a source at the end of the river with the highest elevation, and a sink at the other end (see section 2.7). For an accurate simulation, care should be given to add and subtract approximately equal amounts of water so the river doesn't flood the banks or dry up.

4.2 The Vehicle Model

The simulation involves a vehicle driving across a ribbon bridge. The vehicle has been modeled to allow the user to control the vehicle's velocity and direction using the mouse and keyboard (instructions are provided at the end of this chapter).

The vehicle's orientation is determined much the same way as for a floating object (see section 3.1). The height of the terrain at each of the four corners of the vehicle is determined and used to determine the normal to the vehicle. This normal is used to compute the angles of rotation needed to orient the vehicle.

The height of the terrain at the corners of the vehicle is calculated in the following manner. First, the terrain polygon in which the corner lies is used to determine the equation of the plane in which the polygon lies. Then, the coefficients of the equation and the x and y coordinates of the point are used to determine the z value (altitude) of the point.

The position of the vehicle must also be computed for each iteration. The heading of the vehicle is always known. It is controlled by the mouse, and is simply incremented or decremented according to user input. The same is true of the vehicle's velocity. Thus, the change in position of the vehicle in the x direction is obtained by multiplying the x component of the velocity by the cosine of the heading angle. Similarly, the change in position of the vehicle in the y direction is obtained by multiplying the y component of the velocity by the sine of the heading angle. Adding these values to their respective coordinate values yields the new position of the vehicle.

4.3 Modeling the Ribbon Bridge

The ribbon bridge is made of modular components. The orientation of these components will change as a vehicle drives across them. While they must be allowed to bob up and down and rock back and forth in the water, they must also be constrained in some manner so that the edges of one component match the edges of its neighboring components. This creates a smooth surface over which a vehicle can drive.

This effort requires an approach to physical modeling. The bridge is modeled as having very strong springs between its sections.

This imposes constraints which cause the bridge to act in the appropriate manner, and supplies the orientation for the bridge sections.

As a vehicle drives across the bridge, the weight of the vehicle causes bridge sections to sink in the water, causing ripples. The methods described in chapter 3 are used for water displacement and wave generation.

4.3.1 The Penalty Method

The penalty method approach to physical modeling [Barr 89] is used in modeling the ribbon bridge. This approach forces objects to behave in a specified manner by imposing penalties forces to objects if they behave inappropriately. The penalties become stronger as an object deviates further from the desired behavior.

The motion modeled by the penalty method will continue with no loss of energy (indeed, penalty forces may introduce spurious energy). Thus, a ball will bounce and springs will oscillate indefinitely. To counteract this effect, damping factors are introduced. However, obtaining the interaction between the time slice interval and the damping factors which produces a realistic visual effect often takes some trial and error.

4.3.2 Exponential Springs

Using springs to constrain the sections of the bridge to meet together provides the flexibility needed to simulate the ribbon bridge. However, the springs are not strong enough to hold the pieces closely together. The result leaves large gaps between the bridge sections and the orientation of the sections look more like a parabola than a horizontal line (see fig. 7).

McKenna and Zeltzer describe a modification to the usual spring method which results in a much more effective spring [McKenna 90]. Instead of the usual spring equation:

$$F = kx \qquad \text{(eq. 21)}$$

they introduce the following exponential equation:

$$F = \alpha(e^{\beta|x|}-1) \qquad \text{(eq. 22)}$$

where x is the displacement of the spring from its rest position, α controls the linear strength, and β controls the exponential rise. From these equations, it is obvious that the latter provides a much stronger restorative force as the spring's length increases. In terms of the ribbon bridge, this will hold the sections tightly together.

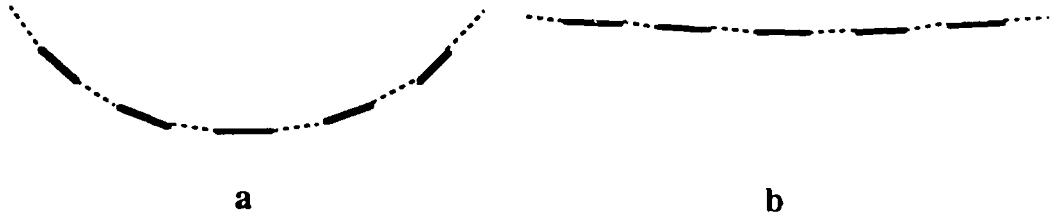


Figure 7. Simulation of sticks being held together by linear springs (a) and exponential springs (b).

The following sections describe the implementation of the penalty method approach to the exponential spring model as applied to a two dimensional chain of sticks. A stick has equal weights on each end, with no mass in between. The model consists of four parts: position, penalty forces, acceleration of the center of mass, and angular acceleration. This is the method used to model the ribbon bridge.

4.3.2.1 Position

Each stick has two ends, A and B, and a center, C. With radius, r, the positional equations for the ith stick are

$$A_i = (C_{ix} - r * \text{cosine}(\Theta_i), C_{iy} - r * \text{sine}(\Theta_i)) \qquad \text{(eq. 23)}$$

$$B_i = (C_{ix} + r * \text{cosine}(\Theta_i), C_{iy} + r * \text{sine}(\Theta_i)) \quad \text{(eq. 24)}$$

The angle Θ_i is initially zero, and is calculated in section 4.3.2.4. Also, the radial vectors, R_a and R_b, for the ith stick can be calculated with the following equations

$$R_{a_i} = A_i - C_i \quad \text{(eq. 25)}$$

$$R_{b_i} = B_i - C_i \quad \text{(eq. 26)}$$

These will be used to calculate the angular acceleration of the sticks.

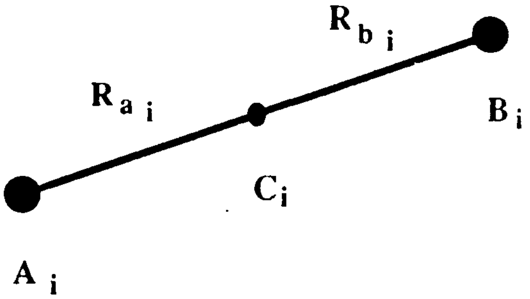


Figure 8. Stick i in the chain

4.3.2.2 Penalty Forces

The spring equation (eq. 22) can be stated in terms compatible with the model as follows

$$F_{a_i} = k1 * (e^{|k2} * (B_{i-1} - A_i)| - 1) \quad \text{(eq. 27)}$$

$$F_{b_i} = k1 * (e^{|k2} * (A_{i+1} - B_i)| - 1) \quad \text{(eq. 28)}$$

The constant k2 is used to limit the force supplied by the exponential. The value of k2 should be between 0 and 1, but experience has shown that the upper range still causes too much restorative force, causing the model to diverge rather quickly. A good amount of experimentation is often needed to determine constant values which produce the appropriate effect.

Damping forces are needed to keep the springs from oscillating indefinitely. An effective method of damping uses a force proportional to the velocity of the ends of the sticks, A' and B', where

$$A'(n) = (A(n) - A(n-1)) / dt \qquad \text{(eq. 29)}$$

$$B'\ (n) = (B(n) - B(n-1)) / dt \qquad \text{(eq. 30)}$$

for the nth iteration and time slice, dt.

Thus, equations 26 & 27 become

$$F_{a_i} = k1 * (e^{|k2} * (B_{i-1} - A_i)| - 1) - k3 * A'_i\ (- 2m_i g)^{\dagger}$$

$$F_{b_i} = k1 * (e^{|k2} * (A_{i+1} - B_i)| - 1) - k3 * B'_i\ (- 2m_i g)^{\dagger}$$

† used only in for computing force in y direction

(eqs. 31 & 32)

where $2m_i$ is the mass of the ith stick.

4.3.2.3 Acceleration of the Center of Mass

Newton's Second Law of Motion (F = ma) can be used to calculate the acceleration of the center of mass, C''. The forces are calculated in the last section, and the masses of the sticks are known. The equation is

$$C''_i = (F_{a_i} + F_{b_i}) / 2m_i \qquad \text{(eq. 33)}$$

This equation can then be integrated with respect to time to determine the position of the center of mass

$$C'_i(n) = C'_i(n-1) + C''_i(n) * dt \qquad \text{(eq. 34)}$$

$$C_i(n) = C_i(n-1) + C'_i(n) * dt \qquad \text{(eq. 35)}$$

Here, n is the number of the iteration, and dt is the time slice.

4.3.2.4 Angular Acceleration

By analogy with Newton's Second Law of Motion, we have

$$T = 2m_i r^2 * \Theta'' \qquad \text{(eq. 36)}$$

Here, T is torque, $2m_i r^2$ is the moment of inertia, and Θ'' is the angular acceleration.

The torque, T, is computed by crossing the radial vectors and forces, yielding the following equation for angular acceleration:

$$\Theta''_i = ((Ra_i \ X \ Fa_i) + (Rb_i \ X \ Fb_i)) \ / \ 2m_i r^2 \qquad \text{(eq. 37)}$$

As in the previous section, Θ can be derived from integrating the angular acceleration with respect to time

$$\Theta'_i(n) = \Theta'_i(n-1) + \Theta''_i(n) * dt \qquad \text{(eq. 38)}$$
$$\Theta_i(n) = \Theta_i(n-1) + \Theta'_i(n) * dt \qquad \text{(eq. 39)}$$

for the nth iteration.

4.4 Driving Across the Bridge

The simulation of the vehicle driving across the bridge involves several processes. First, the location of the vehicle must be detected as being on the bridge, and the section of the bridge it is on must be computed. The vehicle's weight is added to the section(s) of the bridge which supports it. The exponential spring model is then used to get the orientation and position of the bridge sections. The amount of water displaced by the additional weight is computed, and waves are generated to express the effect of the vehicle's weight on the system. As the vehicle drives across the bridge, the amount of water displaced does not change, but is shifted from section to section to provide realism. When the vehicle leaves the bridge, the volume of water it displaced is subtracted, lowering the water level back to its original state.

4.4.1 Car on Bridge Detection

Determining whether or not the car is on the bridge, as well as which section it is on, turns out to be a simple problem. The bridge is initially drawn about the origin, then is rotated and translated to its position on the terrain. Since the angle of rotation and the translation coordinates of the bridge are known, they can be applied to the vehicle to find its position relative to the bridge when the bridge is at the origin.

All that remains is a simple bounds check on the extents of the bridge and the car to determine if they intersect (actually, the centers of the front and rear of the vehicle are used). If the vehicle is on the bridge, it is again a simple check along the x-direction to determine which section the vehicle is on.

4.4.2 Bridge Simulation

For a vehicle to drive across the bridge, it must somehow recognize the bridge as being a surface which can be driven across (as opposed to the water surface, which cannot be driven across). An easy, although memory expensive, way to accomplish this is to keep a separate copy of the terrain database for the vehicle and modify it so that the bridge is considered part of the terrain.

To accomplish this, an array of the (x,y) coordinates which make up the surface of a bridge section is computed and stored for each section. These arrays are used to adjust the vehicle's terrain to reflect the surface of the bridge so that the car can drive across.

As a vehicle drives across the bridge, its weight is added to the appropriate "stick" in the exponential spring model. The slope of each "stick" can be obtained using its endpoints. The slopes can then be used to determine a height value for every point on the bridge using the surface arrays described above. These height values are used to adjust the vehicle's terrain database to reflect the surface of the bridge. The informal formula for this calculation is

vehTerrain[x][y] = bridgeZ - x * slope (eq. 40)

where bridgeZ is the current height of the bridge after water displacement but before orientation, and slope is the slope of the two dimensional "stick" which corresponds to a given section of the bridge.

This representation looks slightly unrealistic when more than one vehicle crosses the bridge at a time. Since the spring model is anchored at each end, the section of bridge between two crossing vehicles remains flat, while the space between two vehicles on an actual ribbon bridge would rise up due to the buoyant force of the water (see figure 9). A possible remedy may be to dynamically change anchor points as vehicles drive across the bridge.

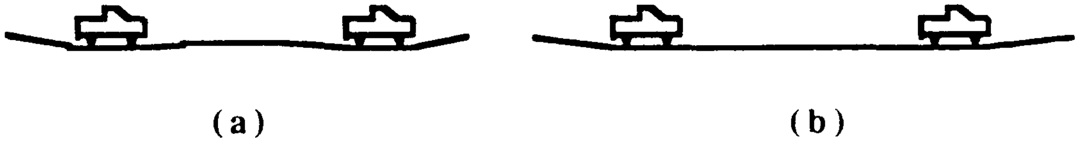


Figure 9. Illustration of two vehicles crossing an actual ribbon bridge (a) and the simulated ribbon bridge (b).

4.4.3 Water Displacement

As a vehic!e drives onto a section of the bridge, that section has weight added to it, which causes it to sink into the water. As the section lowers, it displaces water. In the simulation, the water is not actually displaced, but the volume of the water that should be displaced is calculated and added to the overall volume of water.

Likewise, this volume is subtracted from the overall volume of water when the vehicle leaves the bridge.

This is accomplished in a manner similar to modeling the bridge surface in that an array of coordinates is computed for each section of the bridge. Only this time, the coordinates are of the rows adjacent to either side of the section. These arrays will be referred to as "water arrays".

As the vehicle drives onto a section, the volume of water displaced by the weight of the vehicle is calculated. This volume is added to the rows along both sides of the section, as opposed to the entire volume of water, using the coordinates in the water array for that section. Similarly, as the vehicle leaves a section, the displaced volume of water is subtracted from the water array for that section, giving the effect of the section rising back up to its original position.

Both the front and back of the vehicle are used to create this effect. Thus, if the vehicle is positioned at the point at which two bridge sections meet, both of the sections will have weight on them and both will displace water. Therefore, as the vehicle crosses the bridge, there is a somewhat fluid motion of the bridge sections lowering and rising, with ripples in the water being created at the sides of the sections which are being driven onto or off of, which is the goal of the simulation.

4.5 Controlling the Simulation

The name of the executable file is **bridge**. After typing this the user will be prompted for an input file. The name of the terrain database file is **terra**. Typing this will bring up the demonstration, which consists of a piece of terrain with a river on it, a ribbon bridge across the river, and a small red vehicle. The vehicle can be controlled using the following buttons and keys:

Mouse buttons:

left button	- turn the vehicle left
right button	- turn the vehicle right
middle button	- toggle forward, reverse, and stop

Keys:

"a" key	- increase the speed of the vehicle
"s" key	- decrease the speed of the vehicle
ESC key	- quit

The vehicle can be driven across the terrain and over the bridge. If driven into the river, the vehicle will follow the terrain and will not be effected by the water.

Chapter 5
The Distributed Database Management System (DDBMS)

Work is currently being done at IST/UCF on a distributed database system and network protocol [Moshell 91b]. This method proposes to decrease the amount of traffic across the network, compared to schemes using a single common database, by dividing the terrain database into smaller portions which are distributed among the simulators on the network. The implementation of this approach is explained below, with some ideas as to how boundary conditions might be handled.

5.1 Functionality

Each workstation in the DDBMS can be thought of as being composed of two parts: a **simulator** and a **display**. Simulators and displays on the same workstation can communicate locally, with no network traffic. Simulators can communicate with other simulators and displays by sending mail messages over the network. Similarly, displays can request information from simulators residing on different machines through messages. Displays do not communicate with other displays.

5.1.1 The Simulator

The simulators each hold a portion of the terrain, which collectively form the entire database. The simulators are also tasked with running the models of entities in the simulation. If an entity moves off the section of terrain it is on, a message is broadcast to the other simulators to determine which one holds the portion of the database the entity is moving onto. The entity is then passed to that simulator.

When an entity modifies the terrain database, the changes are recorded locally, greatly reducing network traffic. Still, some network messages may be generated in order to update any displays (see below) which may be holding copies of that portion of the terrain. Messages are also generated by the simulator when user input causes a change in an entity's state. This is more fully explained in the next section.

5.1.2 The Display

The display is responsible for providing the user a window into the simulation. It can be thought of as being attached to a particular entity, usually the one associated with the workstation at startup. Unlike simulators, which pass entities back and forth, the display remains with one entity throughout the simulation.

The display's database consists of a fixed area surrounding the entity with which the display is associated. As the entity moves through the database, portions of terrain are swapped in and out to keep the area in the database a constant size. This will involve mail messages across the network to request new terrain information if the information is not stored locally in the simulator's database. The terrain request is sent to all simulators, and the one which has the information will send a message to the display, which updates its database. The display also receives terrain update information from simulators when terrain in the display database is modified by an entity.

The display must also show other entities which are on the terrain held in its database. To further reduce network traffic, the display runs a dead-reckoned model (or "ghost") of all entities on its database [Moshell 91c]. Instead of having a constant stream of information about the entities on the display database from the simulators which are running them, the display need only be informed of changes in the entity's state (velocity, heading, etc.). Otherwise, the dead-reckoned model can be run locally, with no network traffic.

5.2 Partitioning Strategy

As previously stated, the terrain database is partitioned and divided among the simulators participating in the simulation. The **partitioning strategy** determines how the database will be distributed.

The terrain database is composed of planar polygons. These polygons are grouped together to form **patches**, which are used to reference databases. Patches are grouped together to form **blocks**. These blocks are then distributed among the simulators. With **coarse** partitioning, the number of terrain blocks approaches the number of simulators. **Fine** partitioning occurs when the number of terrain blocks approaches the number of terrain patches (see figure 10).

s1	s1	s2	s2	s1	s1	s2	s2	s1	s1	s2	s2
s1	s1	s2	s2	s1	s1	s2	s2	s1	s1	s2	s2
s3	s3	s4	s4	s3	s3	s4	s4	s3	s3	s4	s4
s3	s3	s4	s4	s3	s3	s4	s4	s3	s3	s4	s4
s1	s1	s2	s2	s1	s1	s2	s2	s1	s1	s2	s2
s1	s1	s2	s2	s1	s1	s2	s2	s1	s1	s2	s2
s3	s3	s4	s4	s3	s3	s4	s4	s3	s3	s4	s4
s3	s3	s4	s4	s3	s3	s4	s4	s3	s3	s4	s4
s1	s1	s2	s2	s1	s1	s2	s2	s1	s1	s2	s2
s1	s1	s2	s2	s1	s1	s2	s2	s1	s1	s2	s2
s3	s3	s4	s4	s3	s3	s4	s4	s3	s3	s4	s4
s3	s3	s4	s4	s3	s3	s4	s4	s3	s3	s4	s4

(a)

s1	s1	s1	s1	s1	s1	s2	s2	s2	s2	s2	s2
s1	s1	s1	s1	s1	s1	s2	s2	s2	s2	s2	s2
s1	s1	s1	s1	s1	s1	s2	s2	s2	s2	s2	s2
s1	s1	s1	s1	s1	s1	s2	s2	s2	s2	s2	s2
s1	s1	s1	s1	s1	s1	s2	s2	s2	s2	s2	s2
s1	s1	s1	s1	s1	s1	s2	s2	s2	s2	s2	s2
s3	s3	s3	s3	s3	s3	s4	s4	s4	s4	s4	s4
s3	s3	s3	s3	s3	s3	s4	s4	s4	s4	s4	s4
s3	s3	s3	s3	s3	s3	s4	s4	s4	s4	s4	s4
s3	s3	s3	s3	s3	s3	s4	s4	s4	s4	s4	s4
s3	s3	s3	s3	s3	s3	s4	s4	s4	s4	s4	s4
s3	s3	s3	s3	s3	s3	s4	s4	s4	s4	s4	s4

(c)

s1	s1	s1	s1	s1	s1	s2	s2	s2	s2	s2	s2
s1	s1	s1	s1	s1	s1	s2	s2	s2	s2	s2	s2
s1	s1	s1	s1	s1	s1	s2	s2	s2	s2	s2	s2
s3	s3	s3	s3	s3	s3	s4	s4	s4	s4	s4	s4
s3	s3	s3	s3	s3	s3	s4	s4	s4	s4	s4	s4
s3	s3	s3	s3	s3	s3	s4	s4	s4	s4	s4	s4
s1	s1	s1	s1	s1	s1	s2	s2	s2	s2	s2	s2
s1	s1	s1	s1	s1	s1	s2	s2	s2	s2	s2	s2
s1	s1	s1	s1	s1	s1	s2	s2	s2	s2	s2	s2
s3	s3	s3	s3	s3	s3	s4	s4	s4	s4	s4	s4
s3	s3	s3	s3	s3	s3	s4	s4	s4	s4	s4	s4
s3	s3	s3	s3	s3	s3	s4	s4	s4	s4	s4	s4

(b)

Examples of different terrain partitioning strategies over four simulators: (a) Fine, (b) Medium, (c) Coarse.

Figure 10.

The advantage to coarse partitioning is that it generates very little network traffic. The terrain blocks are large enough that their boundaries are not often crossed. The disadvantage is that for the entities to interact, they will often all be on the same terrain block. This means that one simulator would run all of the entities, while the other simulators remain mostly idle. The opposite is true of fine partitioning. Here, the work of the simulation is more evenly distributed at the cost of increased network traffic. With the partitioning strategy, the block size can be varied to determine the size which gives the best overall performance.

5.3 Crossing Boundaries

As previously mentioned, entities in the simulation are capable of altering the terrain database. Entities are passed between simulators according to which terrain block they are on. Thus, terrain modifications are made locally. However, there are times when an entity in one terrain block causes a modification to an adjacent terrain block. Consider a bulldozer with its blade down digging a ditch across a terrain block boundary, a tank firing a shell which creates a crater in another terrain block, or water flowing across adjacent terrain blocks.

There are many ways to handle such situations. Several thought experiments are discussed below, using the above examples to give indications of the strengths and weaknesses of each strategy. Of the three examples, water flow presents the most difficulty when dealing with distributed databases.

Before discussing these strategies, two terms must be explained. *Data redundancy* occurs when duplicate information exists. *Data consistency* deals with keeping redundant information. A conflict occurs when two entities are modifying the same data at the same time. If the modifications are handled sequentially, those of the second entity can overwrite those of the first entity. Also, if two entities each hold their own copy of the data (data redundancy) and each modifies the same area, the information must be updated correctly, not simply overwritten, for the data to be consistent.

5.3.1 Strategy 1: Absolute Boundaries

The easiest and most obvious strategy for dealing with modifications to adjacent terrain blocks is to allow modifications only on the block on which the entity is located. This strategy, however, produces the least realistic simulation. Tank fire will produce no craters when shells land in a different terrain block. A bulldozer digging along the boundary between two terrain blocks will make modifications recognized in only one of the two blocks (see figure 11). Likewise, water flow will be stopped at a terrain block border by an "invisible wall."

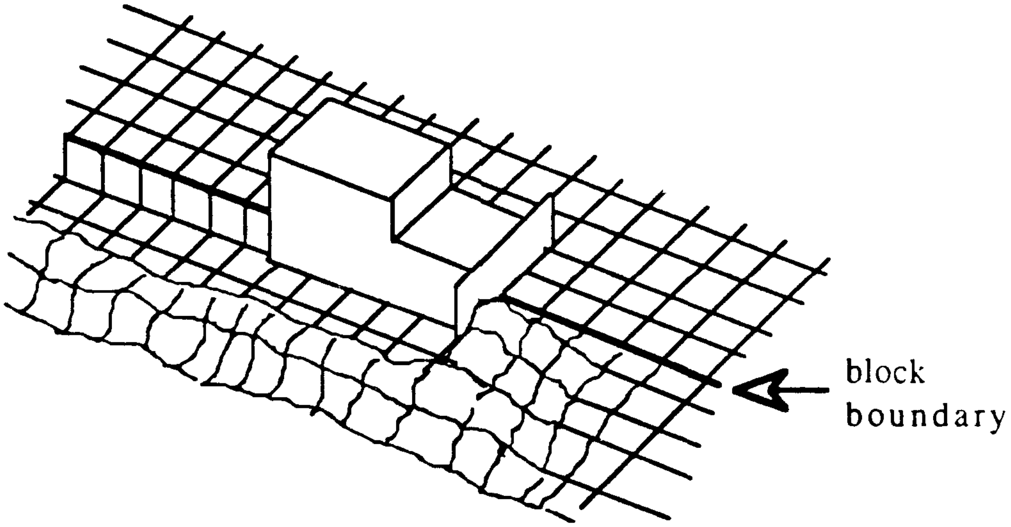


Figure 10: Bulldozer digging along block boundary with absolute boundaries strategy.

The advantages to this strategy are that there is no data redundancy, no problems with data consistency, and no added network traffic. However, this strategy clearly should not be used for a realistic simulation.

5.3.2 Strategy 2: Guess

A different strategy is to make a guess about what the terrain on an adjacent terrain block looks like based upon what the terrain around the entity in the local block looks like. Modifications to the conjectural terrain are recorded and sent to the simulator which holds the actual terrain block, where they can be realized.

The point at which to send the modifications is flexible. It could be after some number of time intervals, or all changes could be stored until the entity has crossed or retreated from the terrain block boundary.

A disadvantage of this method is that it will not produce accurate results, only approximations. The best results will come for entities which modify the terrain immediately around them, such as the bulldozer, since very little of the terrain must be approximated. This method will probably not work well with artillery fire, since the cratering occurs at a considerable distance from the entity. Water flow presents a problem, in that modifications can spread across the terrain block which resides on a different machine. While the flow close to the boundary may be fairly accurate, the accuracy will decrease as the flow spreads.

Other disadvantages of this strategy are the lack of data consistency and a slight increase in network traffic. The advantage of this method is that there is no data redundancy.

5.3.3 Strategy 3: Dual Existence

A third strategy is to temporarily run the entity crossing the terrain block border on both simulators. All terrain modifications are local, producing no network traffic.

This method works particularly well with artillery fire, although special attention must be given to craters formed *on* the border between terrain blocks. This method also works well with bulldozers, although it can produce some incorrect results when the bulldozer is entirely on one terrain block but is close enough to the boundary that dirt piles up across the border. This strategy is not well suited for water. Flow depends upon what is occurring over the entire volume of fluid. Separating the fluid into two parts will produce inaccurate results.

The advantages of this method are that there are no problems with data consistency and there is no data redundancy.

5.3.4 Strategy 4: Boundary Overlaps

Another strategy is to overlap the boundaries of adjacent terrain blocks. When an entity nears the border of a terrain block, the strip of terrain the entity is on actually resides on two different simulators (see figure 12).

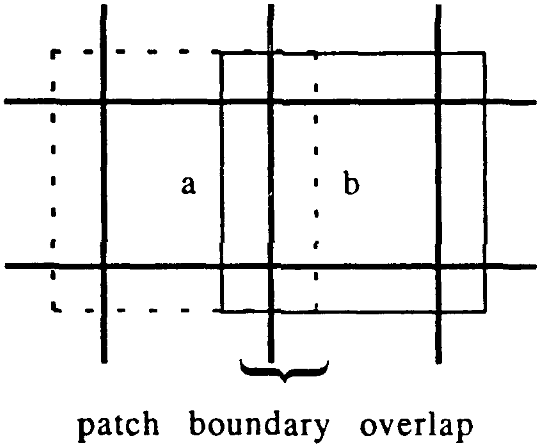


Figure 12: Two patches, a and b, and their overlapping boundaries

This method works well with entities such as bulldozers. All terrain modifications are accurate, although changes on the overlapping strip must be passed between simulators. This method will not work well with artillery fire which crosses over the overlap. No information about the terrain where the artillery fire will land is known. For water, this strategy can work, although there can be data conflict which causes inaccuracies. This occurs because water can flow onto the overlapping strip from both adjacent terrain blocks, as opposed to the bulldozer, which makes modifications in one direction.

The disadvantages of this method are data redundancy, problems with data consistency, some increase in network traffic, and the need for a more complicated partitioning strategy to account for the overlapping borders between terrain blocks.

5.3.5 Strategy 5: Temporary Patch Copies

The final strategy addressed is to temporarily copy sections of a terrain block which reside on a different simulator while an entity is near a block border. Modifications to this temporary section of terrain are made and are copied back to the simulator from which the section was copied when the entity crosses or retreats from the border.

This strategy works well with the bulldozer and tank entities. The modifications are accurate and little network traffic is generated. As for water, this method has some distinct advantages. Water can flow freely across the terrain block boundary in either direction. The flow is accurate. The only problem is that the water flow should reach a stable state before final changes are copied over to the simulator which holds the actual terrain block. This shouldn't create a problem with small flows which will stabilize quickly. But large flows could cause a long delay before the update. A possible alternative could be to send an update every *n* time intervals, which would cause a slight increase in network traffic, but may increase realism.

The disadvantages of this method are redundant data and the problem of data consistency. There is also a slight increase in network traffic. However, the advantage is that this method seems to present a solution which will work reasonably well with all of the simulator entities presented.

5.3.6 Discussion of Strategies

The strategies presented are meant to give the reader an overview of some of the methods considered. Creating temporary patch copies seems to be the most promising approach for implementation in a complete distributed database management system for dynamic terrain. For specific applications or restricted requirements, another strategy may be better suited. Also, a hybrid approach of different methods depending upon the entity may be the best solution.

Chapter 6
Conclusion

This chapter takes a final look at some weaknesses of the work that has been done, some possible extensions and improvements, and some current applications which have been spawned by the work presented in this paper.

6.1 Weaknesses

While the water model chosen for this project provides the required speed and versatility, it is not without some weaknesses. The problems lie more in the implementation rather than the theory, and become visibly noticeable in certain situations.

6.1.1 Directional Artifacts

The partial differential equations for the fluid model are solved in two sub-iterations when dealing with three dimensions. The fluid height field is first computed in the x-direction, and that solution is used to calculate the height field in the y-direction. This produces small inaccuracies in the height field values.

The effects of computing the height field in alternating directions is hardly noticeable with flowing water. They become more clearly evident when dealing with floating bodies. As weight is added to a floating body, additional water is added around the object to make waves and account for the displaced volume of water. If a lot of weight is added to the object, or many small amounts are added in succession, artifacts can be seen extending out from the corners of the object in the x- and y-directions.

This problem may be resolved by determining a way to solve the two sub-iterations described above simultaneously. Methods to accomplish this are being explored, but have not yet progressed far enough to be reported at this time.

6.1.2 Volume Conservation

Section 2.6 briefly describes how the simplifications to the shallow water equations can produce an excess volume of water during an iteration. The height field must be adjusted to account for the excess volume. This is performed by reducing the new volume uniformly over the areas which contain water.

This solution can create disturbing effects in certain situations. Consider terrain with two ponds on it. If a hole is poked in the side of one pond, the water begins to flow out and the volume conservation scheme is applied. As the flowing water spreads, the likelihood of the iteration creating an excess volume increases, as does the amount of excess volume. To compensate, the total volume of water is adjusted, which results in a lowering of the water level in the untouched pond.

At first this may seem to be a minor obstacle. Simply perform a different volume conservation pass over each body of water so that the volume conservation for one pool cannot effect any other pool. The problem is that pools do not remain separated. If the terrain has many contours, as it often does, the pools will merge and separate as the water flows over the land. Keeping track of which pools from the previous iteration should be used to conserve volume with the pools of the current iterations is not an easy problem. This is the major shortcoming of the water model.

6.2 Improvements and Extensions

A number of possible enhancements surfaced as the project progressed. Also, a few additions could be implemented to make the water model more robust. These ideas are briefly discussed below.

6.2.1 Dynamic Computational Boundaries

The fluid model has been implemented over a static square grid. Each iteration, the fluid surface is computed over the each terrain point, whether or not a volume of water exists there. This method is sufficient for the research of this project, but is not feasible for a full scale simulation, where the size of the terrain may be very large.

A possible solution to this problem may be to allow the size of the area over which to compute the fluid surface to expand as the fluid flows. This would require maintaining a square bounding box, or a more complex boundary, which is slightly larger than the area covered by the fluid. This bounding box would be used to compute the fluid height field and for volume conservation.

This approach could still run into trouble if a very large volume of water is involved, such as a river or a breach in a large lake.

6.2.2 Fluid Velocity

Situations may arise where it is necessary to know the velocity of the fluid flow. A possible method for determining the velocity and direction of the flow at each of the surface points may be to compare the volumes of the columns of water across iterations. The fluid height fields of the current and previous iterations are readily available. By comparing the volumes of a column and its neighbors from one iteration to the next, it may be possible to obtain a reasonable approximation of the magnitude and direction of the fluid flow in that column. These values could then be used as needed.

The drawback to this method is that it would probably be very expensive computationally, even for small areas of flow.

6.2.3 Nonrectangular Floating Bodies and Cargo

The method presented for modeling floating bodies applies only to rectangular objects. To make the model more complete, it could be extended to handle objects which are not rectangular. This would not cause a problem for fluid displacement, but the creation of ripples around the object to visually simulate the fluid displacement would be more difficult. However, if the weight of the object remains constant, wave generation may be unnecessary.

Currently, when weight is added to an object, it is assumed that the weight is added at the center of the object. Another extension could be to allow for objects to carry cargo which alters the center of mass of the object. This would affect the orientation of the object at rest, since the center of mass is no longer at the center of the object.

6.2.4 Three Dimensional Springs

The exponential spring model presented in this paper is two dimensional. This means the sections of the ribbon bridge rock back and forth, but not side to side. An obvious improvement would be to extend the model to three dimensions. This would require a slightly more complex spring model, but the concept is basically the same. The most difficult aspect of extending to three dimensions is that it requires the incorporation of an additional torque.

6.3 Applications

Several applications using the water model have been constructed. All of these applications involved other graduate students at the University of Central Florida. These applications are explained below.

6.3.1 The Beaver Pond

The first application constructed using the fluid model involves a dynamic terrain bulldozer, created by Xin Li, which has the ability to alter the terrain in the manner of a real bulldozer. The application has been named the beaver pond. It consists of a square piece of terrain with a pond on it. The bulldozer can be controlled by the user, and can be used to breach a section of the pond wall, allowing the water to flow out.

6.3.2 A Raft on Rapids

Larry Gibbs extended the model of a floating raft to simulate a raft floating down a section of river rapids. He uses the orientation of the raft to derive an acceleration, based on the idea that waves will determine the orientation of the raft and that the raft will move in the direction that the waves are traveling. The larger the tilt of the raft, the greater the acceleration.

Using the acceleration, the velocity and position of the raft can be obtained by integrating with respect to time, the same method used in the exponential spring model. Similarly, the velocity from the previous iteration is used as a damping factor when calculating the acceleration.

This method produces a convincing simulation. The raft moves at different speeds, depending on the water flow, and thus never travels the exact same path twice.

6.3.3 Surfaces

Marty Altman has applied the fluid model to his work with surfaces. He uses the surface points generated by the fluid model as control points for the B-splines used to define a parametric surface. This creates a very smooth water surface.

The disadvantage to this approach is that computing the surface takes a lot of time. The animation is still rather slow using a grid of only 25x25 posts.

An important detail of Marty's model is that he computes the surface from B-splines in the x- and y-directions simultaneously. It may be possible to use this method to solve the water surface height in a similar method, thus solving the problem presented in section 6.1.1.

6.3.4 Rain

Collaboration with both Larry Gibbs and Marty Altman has resulted in simulation of rainfall runoff. However, both attempts failed to produce a realistic simulation. The problem is that the water model works to pull down the higher volumes of water. Thus a single "drop" of water very quickly spreads out, no longer looking like a raindrop. The simulation doesn't look realistic because the size of the terrain is too small. Yet, using a large enough terrain to create a realistic simulation would make the animation painfully slow. Therefore, simulating rainfall runoff which is interesting to watch is not feasible with existing hardware.

6.4 A Final Comment

The fluid model and methods presented in this paper provide a basis for adding water to dynamic terrain. However, a full simulation with dynamic terrain does not yet exist. The reason is speed. Existing hardware cannot perform the vast amount of computations needed to support dynamic terrain. However, the research being done now will aide the creation of a dynamic terrain simulation when sufficient hardware becomes available, and as algorithms and data structures are improved.

Bibliography

[Barr 89]
A. Barr, Teleological Modeling, *SIGGRAPH 89 Physical Modeling Course Notes*, pp. B1-B7, 1989.

[Choquin 89]
J. P. Choquin and S. Huberson, Particles Simulation of Viscous Flow, *Computers and Fluids*, Vol. 17, No. 2, pp. 397-410, 1989.

[De Bremaecker 87]
J. Cl. De Bremaecker, Penalty Solution of the Navier-Stokes Equations, *Computers and Fluids*, Vol. 15, No. 3, pp. 275-280, 1987.

[Fournier 86]
A. Fournier and W. T. Reeves, A Simple Model of Ocean Waves, *Proceedings of SIGGRAPH 86*, pp. 75-84, August 1986.

[FM5-101]
Army Field Manual on Mobility, *FM 5-101*, January 1985.

[FM90-13]
Army Field Manual on Combined Arms River Crossing Operations, *FM 90-13*, February 1990.

[Hua 91]
K. Hua, J. M. Moshell, C. Campbell, and X. Li, Database Management for Dynamic Terrain, *VSL Document 91.22*, November 1991.

[Kass 90]
M. Kass and G. Miller, Rapid, Stable Fluid Dynamics for Computer Graphics, *Proceedings of SIGGRAPH 90*, pp. 49-57, August 1990.

[Max 81]
N. Max, Vectorized Procedural Models for Natural Terrains: Waves and Islands in the Sunset, *Proceedings of SIGGRAPH 81*, pp.317-324, August 1981.

[McKenna 90]
M. McKenna and D. Zeltzer, Dynamic Simulation of Autonomous Legged Locomotion, *Proceedings of SIGGRAPH 90*, pp. 29-38, August 1990.

[Moshell 90]
J. M. Moshell, X. Li, C. E. Hughes, B. Blau, and B. Goldiez, Nap-Of-Earch Flight and the Realtime Simulation of Dynamic Terrain, *Proceedings of SPIE 90*, April 1990.

[Moshell 91a]
J. M. Moshell, in conversation with the author, November 1991.

[Moshell 91b]
J. M. Moshell, R. Dunn-Roberts, B. Blau, C. Lisle, The Virtual Environment Testbed, *VSL Document 91.6*, May 1991.

[Moshell 91c]
J. M. Moshell, C. E. Hughes, B. Blau, X. Li, and R. Dunn-Roberts, Networked Virtual Environments for Simulation and Training, *Proceedings of the 1991 International Simulation Technology Conference*, Orlando FL, October 1991.

[Moshell 92]
J. M. Moshell, X. Li, C. Campbell, K. Hua, and C. Lisle, Dynamic Terrain Databases on Networked Simulators, *Proceedings of the IMAGE VI Conference*, Phoenix AZ, July 1992, to appear.

[Peachy 86]
D. Peachy, Modeling Waves and Surf, *Proceedings of SIGGRAPH 86*, pp. 65-74, August 1986.

[Press 86]
W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, Cambridge, 1986.

[Reece 86]
G. Reece, *Microcomputer Modelling by Finite Differences*, Halsted Press, New York, 1986.

[Sorenson 91]
P. Sorenson, Making Waves, *Computer Graphics World*, pp. 41-46, March 1991.

[Steven 78]
G. P. Steven, On the Application of the Finite Element Method to Problems in Fluid Flow, *Numerical Simulation of Fluid Motion*, pp. 171-197, 1978.

[Stoker 57]
J. J. Stoker, *Water Waves*, Interscience Publishers, New York, 1957.

[Whitham 74]
G. B. Whitham, *Linear and Nonlinear Waves*, Interscience Publishers, New York, 1974.

Appendix I:

A Simplified Model of Soil Slumping

Dynamic Terrain Project
1990-91

TO: Dr. J.M. Moshell

FROM: Julie Carrington, Jennifer Burg

DATE: December 2, 1991

RE: Terrain Relaxation, Project number VSL91.3

One aspect of the problem of how to realistically represent dynamic terrain is the problem of how the terrain behaves when it is dug or stacked up by a bulldozer. Upon being piled up, it should "relax" slightly, i.e., some should trickle down the sides and in general the pile should settle some.

In our program, SMOOTH, we use a variation of an algorithm suggested by P. H. Winston in his book Artificial Intelligence, (Addison-Wesley, 1984, pp. 75-78). He suggests this procedure for making a digital terrain map using a set of altitude readings from just a few accessible places on the terrain using barometer readings which are sparse and noisy and thus have different associated confidence factors. For simplicity's sake, we discuss the two dimensional case; the extension to three dimensions is obvious. Then we have three arrays,

a_i : Altitude at point i. Initial altitudes will be obtained using barometer readings.

b_i : Barometer readings at point i. At most points, no barometer has been read and so b_i is assumed to be zero at those points.

c_i : Confidence factor of barometer reading at point i, between 0 and 1 where 1 means certainty. If there was no reading at point i, c_i is taken to be 0.

In general, the altitudes are weighted averages of the barometer readings and the altitudes of neighboring points where the weights are provided by the confidence factor. Initial altitudes are obtained by:

$$a_i = c_i\, b_i + (1 - c_i)\frac{b_{i+1} + b_{i-1}}{2}$$

After the first iteration, further relaxation is done using neighbors' altitudes from the previous iteration as follows:

$$a_i = c_i\, b_i + (1 - c_i)\frac{a_{i+1} + a_{i-1}}{2}$$

until a point is reached where successive changes are sufficiently small.

In our adaptation of this idea, we have an invisible bulldozer dig up four altitude posts (four is what we consider to be the width of the blade) and stack the amount dug up onto the four posts in front of the bulldozer. Then we relax so that the "dirt" appears to settle.

We found that showing every iteration of that settling process gave an unrealistic appearance. Suppose we have a very tall post (such as one which has just been stacked up by the bulldozer) next to a short one (where we want the dirt to settle onto.) Then on the first iteration, the short post will pull the tall post down to a much lower altitude than even its final settled altitude and similarly, the tall post pulls the short one up too far. On the second iteration, the tall post moves back up nearly to its original position and on the third it is pulled down again, not as far as the first time. In other words, it looks as if it is bouncing. We solved this problem by showing only the peaks so the tall post appears to settle downward gradually. The short one appears to grow suddenly on the first iteration, then it also settles downward. This process looks very realistic.

One goal was to ensure that the volume of dirt does not change in the relaxation process. We found that as long as all of the confidence factors have the same value, this property holds. Note that for purposes of visual realism, this may not be necessary. The dirt before being dug up by the bulldozer may be packed and so after being dug will appear to have more volume.

Ideally, the confidence factor should be some kind of expression which takes into account the dynamics of soil of varying consistencies and conditions. Soil dynamics is beyond the scope of our program which shows simply that a relaxation algorithm can be used to give a more realistic appearance to dynamic terrain.

Appendix J:

Handling Soil and other Bulk Materials

Dynamic Terrain Project
1990-91

Handling Soil and other Bulk Materials with the Virtual Bulldozer

VSL Memo 90.9
25 July 1990
M. Moshell

Abstract

This paper focuses on the bulldozer as a creator of *syntactic structure* on the land's surface. The goals are to minimize the computation required for acceptably realistic realtime soil handling, while leaving behind a rich enough data structure that meteorology, hydrology, forestry, subsequent soil handling and the building of structures are operating on the basis of a reasonably organized set of objects.

A simple class hierarchy of landforms is proposed and described as explicitly as I can in pseudocode. Everything here is intended for implementation within the Virtual Reality Testbed, as augmented with constraint resolution and CurtPhysModeler components, during the fall semester of 1990. Readers unfamiliar with the above should obtain VSL Memo 90.2 from Moshell.

This paper is heavily focused on the idea of knowledge representation for subsequent querying. As such it impacts both database and AI theory, as well as the obvious graphical concerns.

The main goal of this paper is to outline a coherent approach to soil-related problems which will serve as a basis for outdoor versions of Virtual Reality, both in the Army's Dynamic Terrain application and in larger contexts.

CONTENTS

0. The Rationale

Information in the Land. What kinds of information does a piece of land contain?

Depending on the viewer, the answer can be extremely various. A geologist sees evidence of ancient processes; vanished mountains, streams and oceans. A farmer sees written in the vegetation a history of the land's use. A military leader looks for the tactically significant features: cover and trafficability.

- Where can I, or the enemy hide and do battle on this land? Is he therε now?

- How can forces move across this land?

An engineer (military or civil) looks at the hydrology and the soil's stability.

- What structures (roads, bridges, buildings, barriers) can this land support?

- How will the land's usefulness change as time passes and as the weather changes?

The traditional cartographic approach stores two kinds of data: DTED (a regular grid of elevation data points, called "posts") and DFAD (feature data). DFAD consists of either point features ("a church is here") or lineal and areal features (polylines defining the route of a road, the perimeter of a forest, etc.)

DFAD features can and usually do overlap (e.g. a road passing through a forest). To construct a topographic map, contours are constructed from the DTED information and the DFAD information is then registered with the contours. This rather tedious process assures that rivers follow valleys, and other data-consistency issues are resolved.

An essentially similar process is required to produce the polygons used by a realtime image generator. A major additional concern for realtime IG's is the development of *levels of detail* - multiple versions of the database at various resolutions - so that inessential polygons are not drawn when distant regions are viewed.

Landforms and the Grammar of Objects. Our object oriented approach goes slightly beyond DTED and DFAD, to include a third category of data representation: the Landform. Landforms are similar to DFAD areal features, with three crucial differences:

• All the DTED quads (the small quadrilateral regions bounded by elevation posts) know the identity of every Landform to which they belong;

• Landforms, being proper objects, have behaviors. They respond to mechanical and hydraulic actions, weather and the passage of time by modifying the quads under their "care";

• Landforms may only be created in a highly controlled and formal way; we will think of the set of allowable Landforms as constituting a kind of language, and of a particular piece of land as being represented by a document in that language.

The language of DTED and DFAD is conceptually simple; it is simply a textual encoding of marks on a paper map. Nothing in the language itself forbids ridiculous juxtapositions.

The language of Landforms, on the other hand, is intended to reflect a *grammar* of the land. A piece of terrain has a certain innate logic to it, depending on the kind of land it is. A grammar for deserts would be very parsimonious with wooded areas and lakes. If the grammar is a good one, then all user-actions within our simulation system will produce realistic landforms of the given kind.

Aside from producing correct "sentences" or structures, the second purpose of a grammar is to render comprehensible that which is produced. Our Landform grammer will leave behind both a part-whole and a class hierarchy, for subsequent use by any Virtual Entity which needs to undersand and interact with the land.

The Dual Between *ab initio* and *ad hoc* Knowledge Representations. In the world of AI, a classic struggle has taken place. One body of opinion holds that a small collection of "semantic primitives" - objects, actions, attributes - should suffice to describe the world. If physics and chemistry were well enough understood and expressed in the appropriate primitives, then all else could be deduced.

Ab initio: "from first" (principles).

A second body of opinion asserts that most so-called "intelligence" is just the recognition of situations drawn from a huge reportoire of "scripts" and the application of the appropriate behavior, based on the details of the situation. This viewpoint has increasingly carried the day, as workable AI systems are slowly developed.

Ad hoc: "to this" (do whatever it takes to get this specific job done).

Our attempts in early 1989 to construct models of bulldozers in action fell prey to our inadquate understanding of this dichotomy. If you design a simulation that is too *ad hoc* , you get nothing but a bulldozer. If you try to be too *ab initio*, you would build in all of physics, and take four Crays and forever to compute anything.

The formalism I propose below is intended to balance between the extremes. Its goal is to provide acceptably realistic simulations of common soil operations that can be executed in realtime (defined as >= 4 simulation frames per second) with our network of workstations.

The quickest way to explore these concepts is to describe our to-be-built simulation system in action, in storybook form.

1. The Scenario

Definition: berm: a small manmade hill, usually long and narrow.

A small stream flows east to west; the bulldozer is on the north bank (figure 1).

The dozer constructs an arc-shaped tank barrier ditch, with two berms b2, b3, north of the stream (figure 2). The dozer leaves behind its fore-berm b1.

The dozer backs up, turns to the right and pushes its new fore-berm b4 down the hill and into the stream. Two additional berms b5, b6 result. (figure 3)

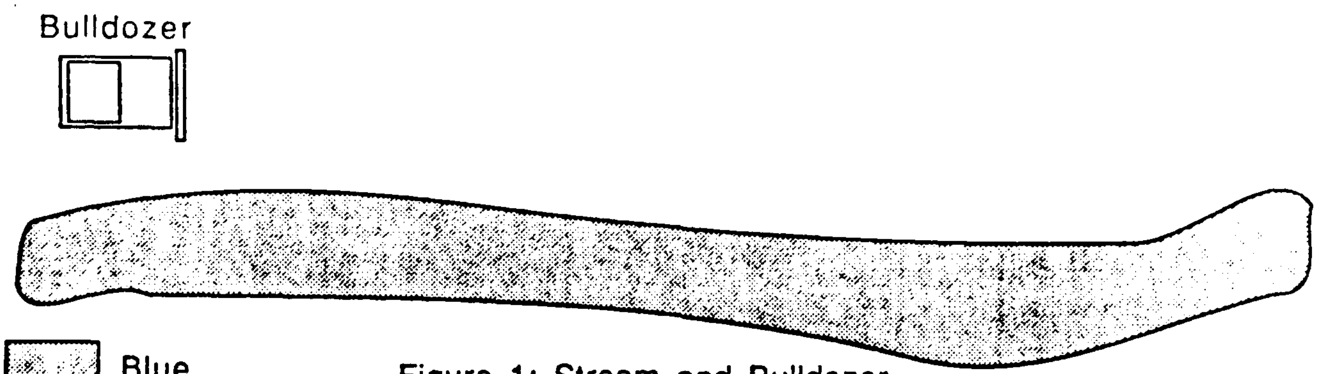


Figure 1: Stream and Bulldozer

▨ Blue

▨ Brown

□ Green

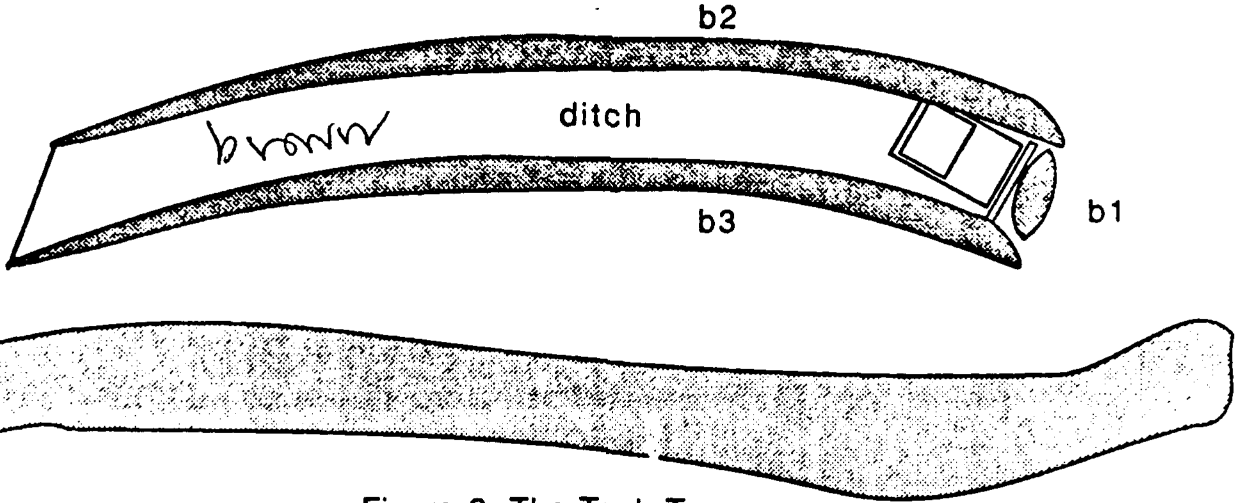


Figure 2: The Tank Trap

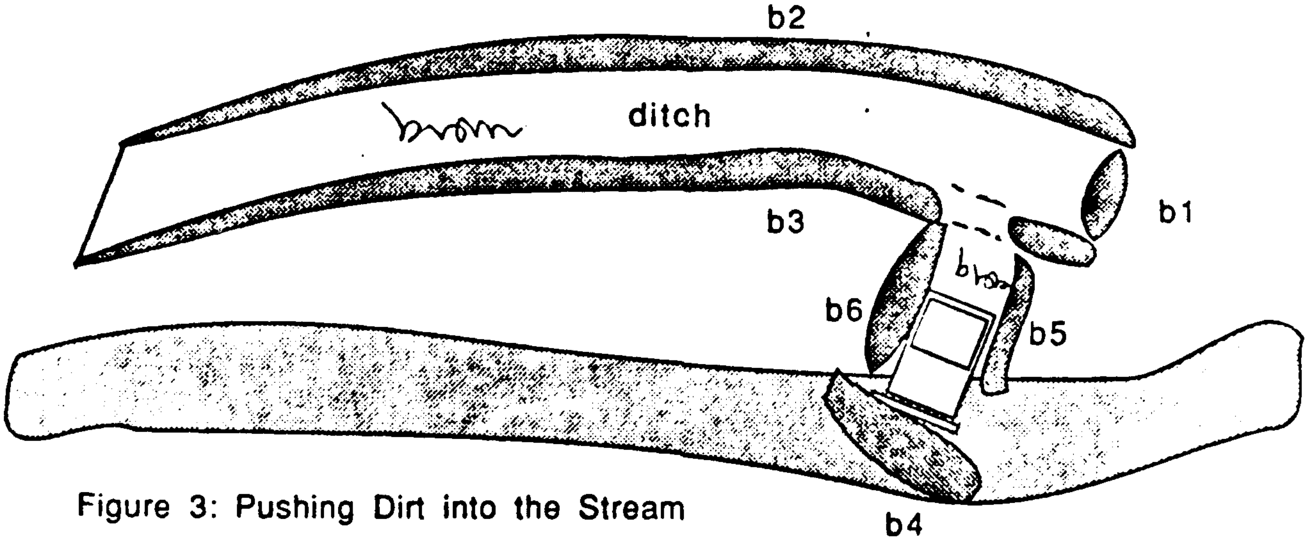


Figure 3: Pushing Dirt into the Stream

-7/26/90- page 4,5 -

The dozer again backs up, turns right and then left, and pushes a second mass of soil (fore-berm b7) into the stream, destroying b6 and creating b8 and b9.

The dozer now leaves. Water in the stream slowly accumulates into a lake, until its height matches the top of the soil dam that has been pushed into the ditch. When that happens, it flows over the top of the dam. (figure 4)

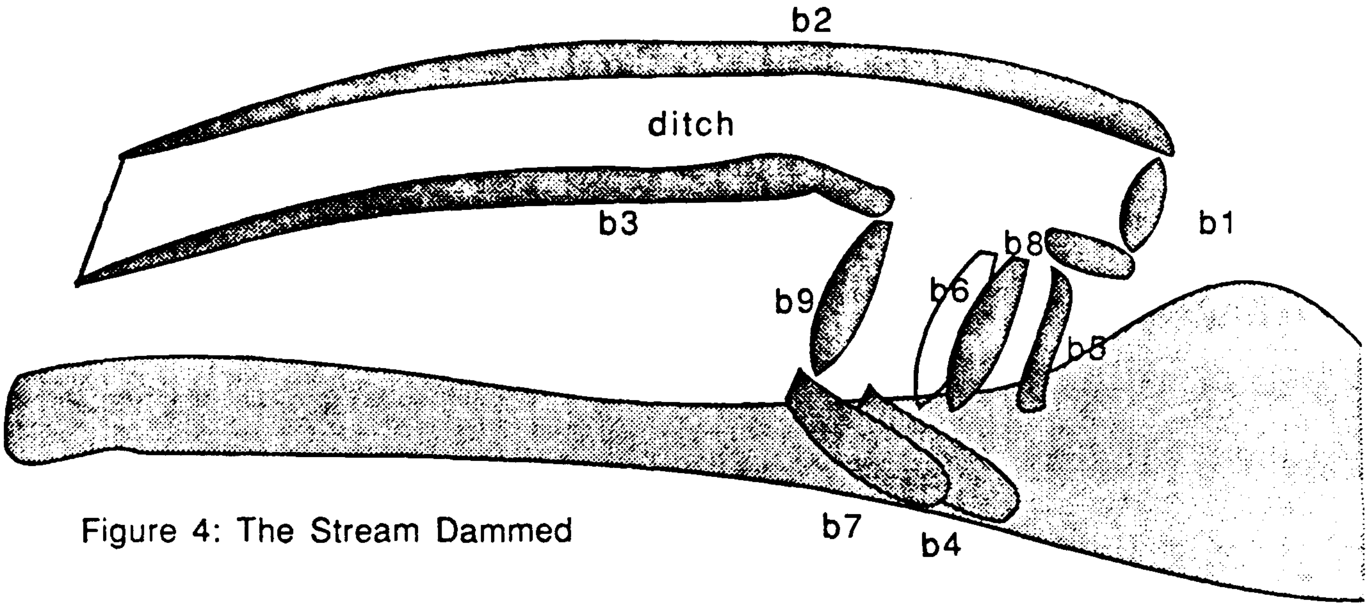


Figure 4: The Stream Dammed

If a tank arrives before water crosses the dam, it can traverse the dam; otherwise, it gets stuck.

If the tank tries to cross the tank barrier in a north-south direction, it can do so UNLESS it has rained. If it rains, the ditch fills with water and becomes impassable.

The prime reason that a scenario like this is interesting is that the underlying simulation system must be functional for an infinite class of similar situations. No matter how you dig the ditch, drive the tank, or get rainfall, things must work reasonably. When this happens, we can fairly claim to have a useful knowledge representation over this restricted domain.

2. The Part-Whole and Class Hierarchies

The class hierarchy described in Figure 5 shows Terrain as falling into two broad subclasses: Landforms and Overlays.

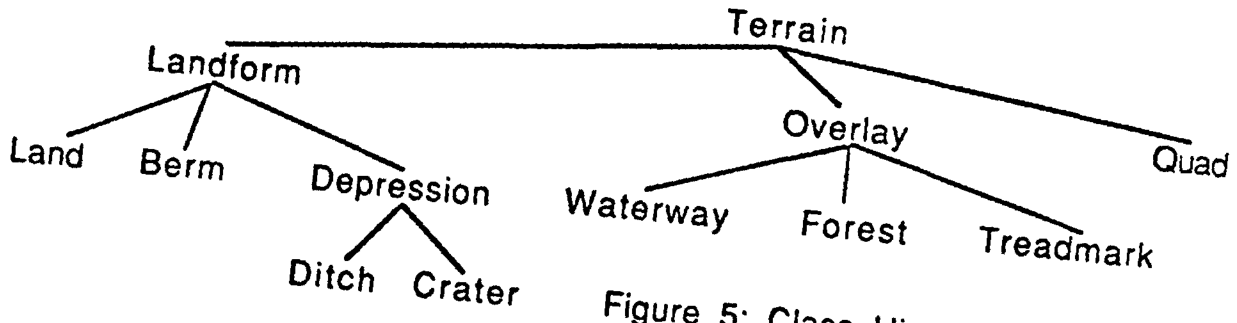


Figure 5: Class Hierarchy for Terrain

Landforms are centrally concerned with topography (the z coordinate); they are a structured (and sometimes redundant) representation of DTED information. A Landform is always a collection of quads (stored in the Landform's QuadList).

Overlays are centrally concerned with attributes; they are the spiritual descendants of DFAD. They have polygonal boundaries, which lay across the landforms. Their boundaries do not necessarily correspond to the discrete structural grid of the quads.

Quads are the basic "atoms" of the terrain. A quad has four edges, parallel to the x and y coordinate axes. Each corner has an elevation value. The quad has several lists:

- a MemberModel list, which names all the movable models presently touching this quad (e.g. a tank, a bridge footer, etc.);

- a MemberLandform list, which names all the Landforms to which it belongs; and

- a MemberOverlay list, which names all the Overlays which pass across it. It also has its own Color and Soiltype attributes.

Quads may vary in size, to allow micro-terrain, terrain relaxation, etc. They are also in general cut into two triangles for viewing purposes, so that planar surfaces are available for shading.

Land is the concrete class of which an instance, named *theLand*, serves as the underlying structure for the entire scenario. *theLand* contains all quads in the simulation.

Berms are the most fully developed objects in this paper. The basic idea is that we want to be able to do bulldozer operations as a direct and simple transaction between five objects: a dozer, three berms and a ditch, rather than (as is now done) the detailed manipulation of elevation posts within theLand by the dozer. How can we do this?

Let's look at a detailed sub-scenario, in figure 6. The berms are deliberately over-simplified (large quads are used) to keep the drawing comprehensible.

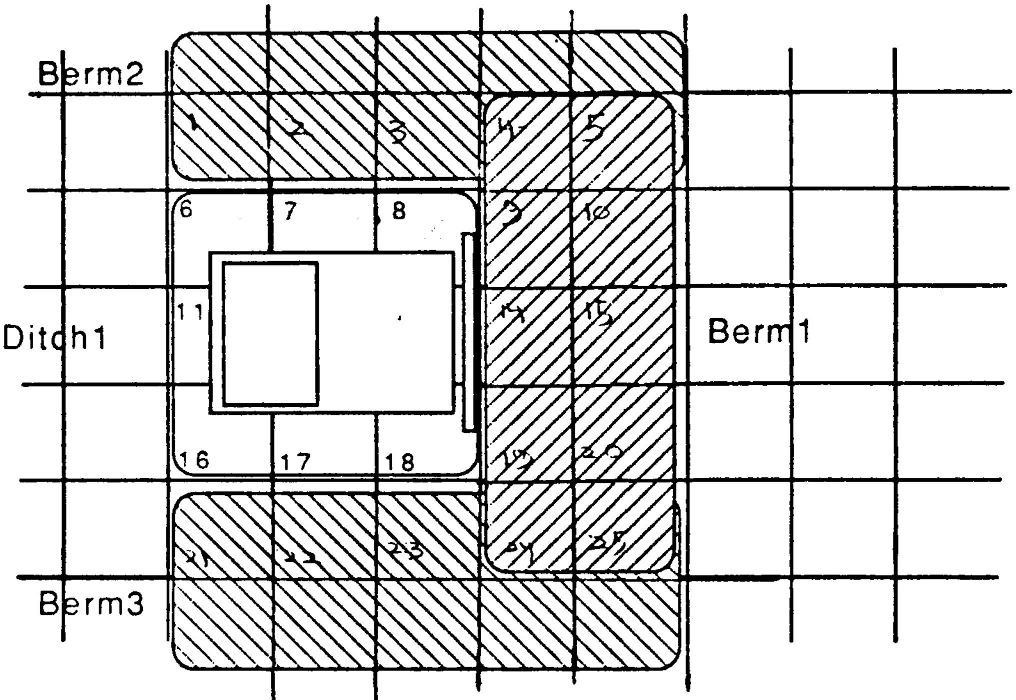


Figure 6: Overlapping Berms and Ditch

All the quads belong to theLand. Quads 1,2,3,4,5 (and five unnumbered quads) also belong to Berm2; Quads 4,5,9,10,14,15,19 and 20 belong to Berm1, etc. The landforms overlap as advertised. Now - what can the dozer, the berms, the ditch and theLand do to one another?

When the dozer is driving across plain land, it need only inform the land of its location (four corners), plus velocity vector. The dozer is responsible for always traversing (with its leading edge, as defined by the velocity vector) every quad which will ever be underneath the dozer. Thus, theLand knows which of its quads are being run over.

If any of the newly-run-over quads contains any moving models, the dozer is informed of that fact. This simplifies initial collision detection (e.g. with models such as trees, rocks, etc) because you don't have to constantly search every model against every other.

Likewise, if any newly-run-over quads belong to any Landforms other than theLand, the dozer is informed. If the blade is up, the dozer doesn't care; it simply interacts with theLand "in bulk" to determine if it can move, or if it's stuck (more later on how this is done).

Blade-down situations. First let's consider the dropping of the blade on plain ground. In figure 5, the blade lands on the border between quads 8,13,18 and 9,14,19. Three berms and a ditch are created immediately. The left berm (Berm2 above) initially consists solely of quads 4 and 5; the fore-berm (Berm1) consists of the ten marked quads; Berm3 is quads 24, 25; Ditch1 has no quads in it yet.

None of these quads is elevated yet; they've just been added to the QuadLists of the newly created berms and ditch, and the berms and ditch have been added to the MemberLandform lists of the quads. Berm1 is distinguished; the dozer knows that Berm1 is in front. Likewise the ditch is known to the dozer.

As the dozer's blade moves forward, the dozer calculates how much volume has been scraped up (this is a direct query to theLand, actually). Messages are sent to the ditch and to Berm1, specifying this volume of dirt and describing the blade's movement in (x,y,z) (probably by sending two x,y pairs specifying the ends of the blade, and the z of its lower edge.)

> This order of business is to avoid having the ditch and Berm1 both compute the volume.

The ditch informs its member quads as to how they should react. It might ignore the volume and simply adjust the z of the traversed quads.

Berm1 has a more interesting task. It must use the incoming volume info and the new (x,y) coords of the blade's ends to compute its own new shape. This will involve some growth during startup, and then the translation of the "full berm" with minor volume adjustments as the blade is raised or lowered.

Then it must compute the volume of dirt that falls off the ends into berms 2 and 3, and tell them. ("3.4 cuMeters of dirt just fell on you at xy location 23,44").

Berms 2 and 3 are now responsible for reshaping themselves on the basis of the new information. That will probably just involve raising the named quads and splining the adjacent quads (quad 3 in this case) to fit. If berm 2 didn't previously own quads 4,5 then it just grew, so it must add them to its QuadList, and itself to their MemberLandform lists.

A berm is also, in general, of a different color than the undisturbed quads. For undisturbed quads, the color (e.g. grass-green) is not dependant on the soiltype, but once the quad has been bulldozed, its color changes to the appropriate color for the soiltype. Even if the berm is subsequently flattened, it will still be brown instead of green (at least until we get to the section on ground cover, below).

In summary, data flows as follows during each simulation step:

(1) Dozer moves forward, asks theLand how much volume it just ate.

(2) theLand replies with a volume. Dozer informs ditch and Berm1.

(3) Ditch and Berm1 redraw themselves. Berm1 tells Berms 2 and 3 what their spillover is.

(4) Berms 2 and 3 redraw themselves.

Whenever the tractor backs up, its current set of berms and ditches is abandoned. When the tractor goes forward, a new set is created (if the blade is down). This proliferation of data structures amounts to trading space for time, as will now be shown.

Economies of Scale. The main advantage of this structure is that none of the above message traffic required any knowledge of the granularity (size of quads). The discussion is all in terms of bulk materials: how much volume, and where in x,y,z space it'll be delivered; where my blade is and where I'm going.

Thus, a viewing system (such as an Iris elsewhere in the net) can be maintaining its polygonal view of the scene at any desired level of detail. The ghost objects corresponding to berms and ditches there will have to include their own splining routines so as to properly "digest" incoming bulk material deliveries. Rather than shipping lots of update-elev-post info over the net, we're just shipping minimal descriptions of the changes to those berms and ditches which changed.

The central database must store an explicit description of each ditch and berm, somehow - perhaps as a log of "loads delivered" - so that a querying remote workstation at a later time (it wasn't "looking at the action" when the earthworks were built) can get the data for rendering its own local version of the berms and ditches.

Alternatively, the central database can just keep track of where within the network a fully rendered version of the terrain exists, at each different level of detail (quad size) and send a querying workstation to the right place to get the elevation posts.

Crossing Old Berms. What happens when the dozer backs up and turns out of its old ditch, as in figures 3 and 4? Not much, really. The polygons in the disturbed region of Berm3 are lowered, because they became part of Ditch2 and Ditch3. They are still listed as members of Berm3 (which doesn't cost us any execution time, just some memory space).

This information (former membership in a berm) could be useful in the future, inasmuch as soil that has been underneath a berm is more compressed than otherwise, and probably damper. The most important fact is just that it doesn't hurt to leave these quads in Berm3's list; purging would be expensive and difficult.

3. Overlays

It is desirable to retain the topographic information in the Landforms even when we wish to superimpose features such as a stream. The topo data will describe the streambed's shape, which is essential when water levels are to rise and fall.

Thus, the "feature data" in our system must drape itself across the landforms. Let's consider the water feature, since I'm looking to Chuck Campbell to build this feature into our simulation as his Master's thesis.

A Waterway has an altitude, a polygonal boundary and a color. It is rendered as a horizontal polygon. With z-buffering, any berms or topo lumps which project above the surface will automatically appear as islands. <more later>

Appendix K:

Object Oriented Databases for Dynamic Terrain

Dynamic Terrain Project
1990-91

Review : Object Oriented Terrain Databases for Visual Simulators

Brian Blau
VSL Document 91.35

Visual Systems Laboratory
Institute for Simulation and Training
University of Central Florida
Orlando, Florida

Introduction

This document is a review of the work done for the Dynamic Terrain Project at the Institute for Simulation and Training during 1989-90. The topic is object oriented databases which provide dynamic terrain capabilities. This project was part a masters thesis. A project report [Blau90] and demonstration programs were presented at IST in January 1990.

Motivation for Object Oriented Terrain Databases

In recent years, the computer revolution has made possible high fidelity computer simulations. Visual simulators have traditionally employed static terrains over which active objects move. In these environments, actions that should alter the terrain sometimes result in visual changes. The construction of earthworks, erosion and traffic damage are examples of the complex effects which occur in real life and need to be emulated in graphical simulations. Unfortunately, existing systems rarely modify the terrain's internal structure and behavioral characteristics.

Developments in hardware and software are now making it possible to manage and display dynamically changing terrain in real-time. Using an object oriented approach, it is now possible to implement a system which supports dynamic terrain. The project described in this report has developed novel data structures and modeling algorithms for land formations that can be modified during display.

Motivation

Massive processing power combined with the need for accurate simulations has created the computer simulation industry. This industry focuses its efforts on how humans use computer inputs and outputs to better train for a particular task. Many private and public institutions conduct research to continually improve the training/response scenario. An example of a training simulator is the SIMNET network of tactical tank trainer. In this simulation, many separate tank units participate in a single computer networked battle, giving the soldier a feeling of being in a large battle. This device helps the U.S. Army give soldiers effective training in a "semi-real" situation so they might be better prepared for actual battle.

Some computer based simulations are used for training, but there are an equal number of computer applications which are not training situations, but have more social implications. Examples can be seen in Walt Disney World and Universal Studios in Florida. The engineers there have created the Star Wars/Body Wars and Hanna Barbara ride experiences. In these similar rides, passengers sit in a room where they view an animated film. The motions portrayed in the film are felt by passengers as a motion platform moves them around. In some of these cases, computers can generate the film shown to the audience as well as coordinate the control of the motion platform.

This project focuses on how to make computers better represent the real world in a simulation and training environment. Although there have been many advances in computer simulations, there still remain many hard problems to solve. The work described below brings into focus some of the difficulties with today's computer simulations.

Data Paths

The data flow in a simulator mimics how a soldier fights in a real battle. First there is some input to the fighting machine, or in the simulator's case, input through discrete switches or analog controls. Next, a weapon is fired and some destruction occurs, the simulator computes the trajectory of the round and it determines if any object has been hit. Finally the soldier sees the result of his action and again inputs some action to his fighting machine. Here the simulator computes the graphical screen and displays it on a monitor. Finally the cycle is complete and input is again introduced to the unit.

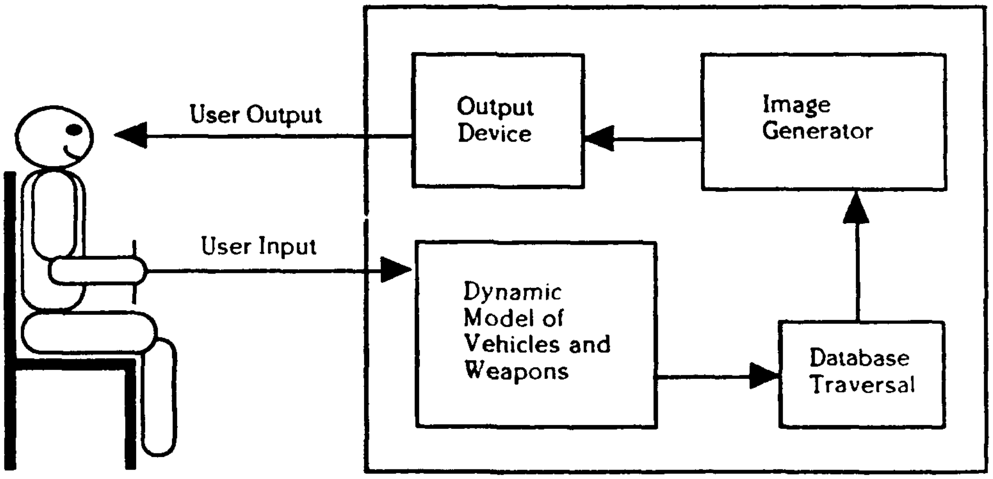


Fig. 1 Life cycle of user input to user output

User input can come in many forms. There can be triggers, knobs, pedals, switches, steering wheels, throttles and buttons. Once the input has been gathered inside the host simulation computer, the computation system starts. Here all of the inputs are converted to world coordinates so all data is in the same coordinate system. The physical dynamics of the simulation determine the velocity, direction and explosive effects of all objects which are in the simulation. Next, a database is read to determine what environment surrounds the user. This data is then sent to the **image generator** and a picture is displayed. There may also be lights and gauges for the user to read.

Visual Systems

One important output of a simulator is the **visual display**. Here the user is presented with some graphical representation of the virtual world which is located on the other side of screen. This display is generated by a high speed **graphics engine**, commonly called an **image generator**. This functional component takes three dimensional information about a scene and using mathematical transformations, projections and rendering routines, produces a picture display. Inputs to the image generator come from the user's position in the virtual world and a database which contains a model of the world. The output of the graphics engine is through a CRT monitor and video projector.

Dynamic Terrain

Even though there are many successful image generators on the market, they are all lacking in one respect. Some of the inputs the user is giving to the simulator unit are not being recognized. These may be very subtle inputs, but they still are not handled. For example, if a tank is driving close to a river bank, but not actually in the river, the ground underneath the tank may be soft. When the tank moves it should leave marks in the soft ground. Also, if a tank fires its weapon and the round does not hit another object, it then strikes the ground leaving a crater. A simulation of these events should correspond to their real world counterparts.

If this capability were to exist in simulators, the functions to create craters and tank tracks would be located in the image generator, or some piece of equipment closely connected to it. But the databases and graphics engines in today's simulators do not have this capability.

Both of the events described above modify the underlying terrain. Previously the terrain was named as a static model because it would never change, but it now must be made a dynamic model. We will now call a dynamic model of terrain, **dynamic terrain**.

There are a few possible solutions to the problem of dynamic terrain. When a tank leaves tracks, it may be enough to simply display a black mark where the tank has driven. This is an attempt to leave a permanent mark of the tank's presence. This method is acceptable when only looking at the tracks, but it is not acceptable when trying to navigate them. This effect can be seen in Fig. 2, Tank A. Here textures are left behind the tank as it moves across the terrain.

If one tank is in pursuit of another, then the visual and motion cues felt by driving over the tracks might be important. After the passage of a column of tanks, a roadway is often impassible to wheeled vehicles. This effect can be seen in Fig. 2, Tank B. As the tank moves across the terrain, changes to the terrain polygons are made automatically. A small depression in the polygon signifies that the ground is lower at these points. The depth of the crater might give the enemy some idea of its size. These visual cues are not part of today's visual systems. The inputs that have been given (e.g. driving over soft ground) are not registered by the simulation computer.

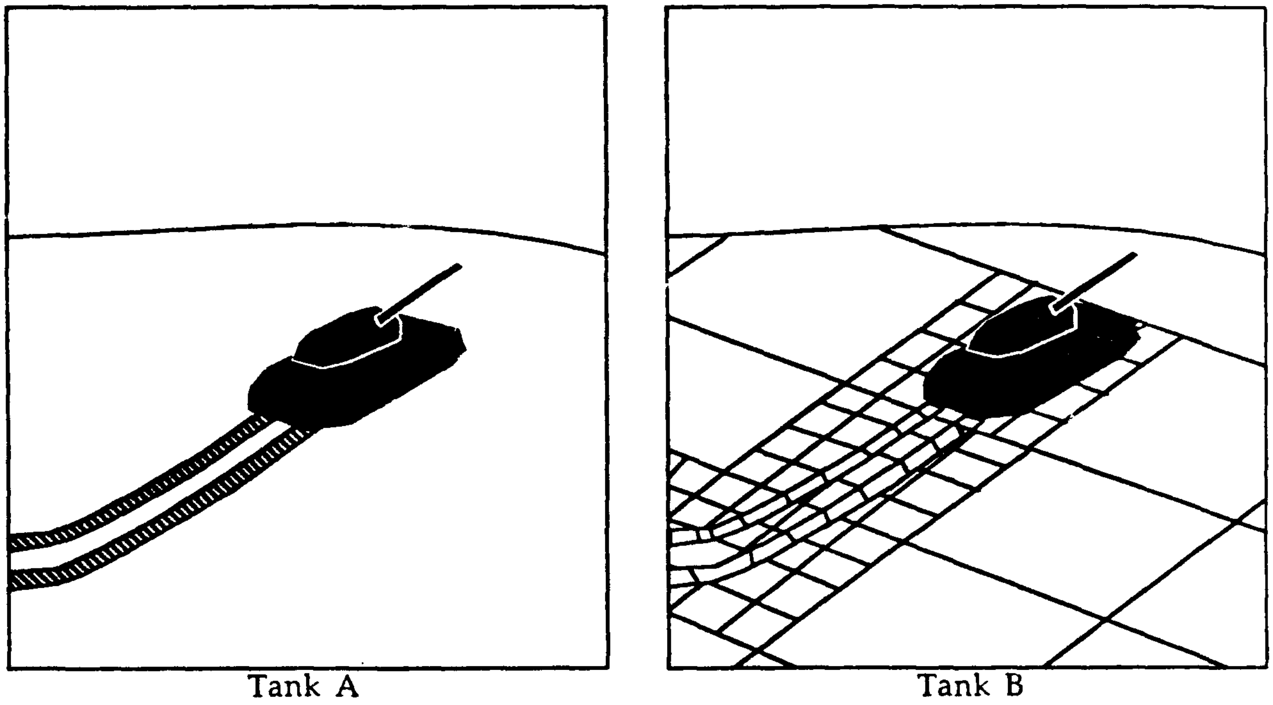


Fig. 2 Tank A may look good, but Tank B leaves real tracks

Not only does the dynamic terrain model apply to terrain, but if the model is extended, all forms of databases will be able to share in the dynamic nature of models. As it stands now, the simulation industry has put a restriction on what can and cannot move in a simulation. By making all models of a simulation dynamic, we can better recreate reality.

Object Oriented Design for Simulations

There has been a software evolution which may hold the answer to some of the problems with visual simulators. **Object oriented technology** has the inherent ability to manipulate and manage the complex data structures which are necessary for dynamic terrain.

Many parts of a simulation can be described as **encapsulated data structures, or objects**. A house, tree or tank can be an object. Objects are instantiations of data structures with **actions** and **slot variables**. The slot variables of an object are objects themselves. Sending a **message** to an object requires that object to respond by executing one of its associated actions. **Actions can be passive or active** and the only way the outside world has access to that object is through one of the actions. The collection of an object's **actions are known as its protocol.**

Because objects are encapsulated data structures, they are dynamic by nature. By knowing the protocol, objects may send messages to each other without regard to implementation or method of execution. Thus when a message is sent to an object and the action is passive (i.e. the action is simply to assign a value to a slot variable) no message propagation takes place, but if the action is active, then many messages may propagate and affect other objects located in the system. Nesting of objects is possible through the use of slot variables. This is called an **object hierarchy** or **part-whole hierarchy**. One slot variable may contain an object, which has a slot containing another object.

A simulator database may be modeled by using objects. Houses, trees, rivers, lakes and terrain are candidates for these encapsulated data structures. A simulation database may easily take advantage of the part-whole hierarchical structure. For example, a house object may have as its parts a door, a kitchen, a fireplace and a room. The kitchen object may in turn have a stove, a sink and a cabinet. The terrain of a simulator database may be modeled the same way. A terrain may be one object, itself comprised of smaller pieces of terrain. Each smaller terrain may contain even smaller terrain objects. A terrain object may even contain houses and trees as parts. This encapsulation of objects gives the terrain a natural hierarchy. Using the object oriented structure to implement the database, both static and dynamic models can be incorporated into the database.

Within an object oriented design, each part of the simulation is independent from the others. Objects may receive messages and act upon them accordingly. Because each object is encapsulated, different parts of the simulation will not know how other parts are implemented. The only communication route is through message sending. This means each object must have a well defined protocol which must be known to all other objects. This is the main restriction on how objects interact, and it is not a drawback, but a feature. Because the implementation is not known to the outside world, there will be no direct modification of the internal variables. This encapsulation leaves the object free to use any implementation without fear of losing reuseability or portability. The state of the object is defined by its internal variables, while modification is done through its standard protocol.

Accomplishments and Lessons Learned

The OOTDB was a proof-of-concept experiment. Because there were many unknown variables at the start of the project, some of the results obtained were unpredictable. In this section, some specific conclusions and recommendations will be presented about the object oriented design, the terrain editor and the use of Smalltalk-80 and GemStone.

Object Oriented Representation of Terrain Geometry

One of the main ideas presented in this project was the use of object oriented technology as a foundation for the construction of the terrain database. Object oriented design was chosen because it facilitated the natural representation of real world data in the form of structured programming.

The decomposition of a region of the world can be described spatially by using the part-whole relationship. It is easy to say that a mountain range has a city and lake located in its area. This straightforward approach to the design of the part-whole hierarchy also is used in the design of the class hierarchy. It is very natural to say that a region of terrain has water formations and man-made formations as its parts. This breakdown of the terrain into its object oriented representation is usually expressed without any difficulty, but an experienced geographer would perhaps be better equipped than a computer scientist to describe realistic terrain attributes.

Complex Queries

Incorporated into this project are methodologies from object oriented databases. It is important to realize that the terrain database is not an implementation of a traditional, query processing data manager. Rather it is a specific collection of classes which can manipulate dynamic terrain databases efficiently.

The query language processor of the terrain database is a part of the Smalltalk-80 language. The database modeler must specify the operational code of the query, and the host system will do the dynamic binding of the query call. This implementation of an object oriented database will support the complex queries which are generated from a dynamic terrain simulation. Queries such as those from a geographic information system and a real-time simulator can be supported by the OOTDB. The work of this project has focused on the efficient representation of spatial data. Therefore more work will be needed to fully demonstrate the real-time capabilities of the OOTDB.

Different Spatial Data Structures

Object oriented design states that the problem statement should be broken down into the base structures to determine the relationships which exist among different the entities. Once these relationships have been established, the common structures are found and placed in classes. These classes become the abstract classes, placed near the root of the class hierarchy. The class hierarchy is then refined and tested, and the design process is repeated until the resulting hierarchy is satisfactory. Using this method, the terrain database was formed for this project. It contains one root class which manages all of the part-whole hierarchy and lets more specific subclasses use these resources.

After looking at the design process used for this project, it became clear that some of the original design could have been done more carefully. This is evident in the spatial data management. If a structure like Antony's [Antony

88, 90] was followed, the performance and modularity might have improved Specifically, a better merger of the class and part-whole hierarchies and spatial relationships would have resulted in better data management. Some design considerations, like the one mentioned above, were discovered late in the implementation of the project. It also became clear that the DCEL structure was a good choice for the spatial relationship, but the quad-tree might have been easier to work with, and most likely, a combination of these two data structures would have been very useful.

Frameworks

One final comment on the design of the terrain comes into light when looking at object oriented design. One of the most important aspects of object oriented design is the natural tendancy to reuse classes. A particular form of reuse is called a **framework** [Johnson 88]. This would give the class structure of the database as much reuseability and modularity as possible. It would also be convenient when distributing the terrain database to users. They would receive a copy of the terrain framework and would only need to know its protocol to be able to use it.

Smalltalk-80

The Smalltalk-80 programming environment was chosen because it was the only available application which met the original design goals. It is available on popular UNIX workstations and is relatively inexpensive. But the use of this tools had some effect on the outcome of the experimental *results in terms of performance.*

One capability of the database is the ability to manipulate large amounts of data. The Smalltalk-80 system is image based, meaning that all source code is stored in binary format in the environment. The image also contains objects and its size is determined by the number of objects currently in use. As the object count increases, the internal memory manager allocates space. Large terrain databases may be slow in processing because of the virtual image, but they are only limited in size by the image size.

The only implementation of the database is in Smalltalk-80. This programming environment is available on very fast UNIX workstations and its performance on those computers is quite good, for some applications. An advantage to using Smalltalk-80 is that the language supports late binding, which means that classes can be added while an application is executing. This is a powerful language mechanism which is available in Smalltalk-80, interpreted rather than compiled language. This price of using an interpreter is performance.

Another disadvantage is Smalltalk-80's lack of popularity among computer and simulation professionals. Because object oriented ideas are new, many commercial manufacturers have not accepted this as a legitimate technology. Implementing this project in Smalltalk-80 meant that many people would be exempt from using it and rehosting would require a large effort. Recent releases of Smalltalk-80 have given this product more exposure to computer professionals. In the future, object oriented programming will become popular in all aspects of computing. Mandates by the government in the from of software standards will give this technology more exposure.

Gemstone

One of the other strengths as well as weaknesses is the software package GemStone Object Oriented Database. The most useful part of this tool is the persistence of Smalltalk-80 objects. It is very easy to use GemStone to permanently store and retrieve objects. Using GemStone in its basic form requires very few commands, and the objects can transfer back and forth between Smalltalk-80 and GemStone easily.

One of the main problems in using GemStone is access time. Even though the workstation which houses the GemStone server has a high processor speed, database access is always slow. Typically, the time to read a small (four patch) terrain database into Smalltalk-80 is about 30 to 60 seconds. This speed would be unacceptable for any real-time simulation.

Another drawback is moving classes from Smalltalk-80 to GemStone. The only way to transfer class definitions to GemStone is by hand. Each Terrain class and its subclasses must be meticulously recreated using the GemStone Class Browser. There is no automatic of doing this transfer.

An Experiment on Performance of the OOTDB

To show that the OOTDB is a viable resource for use in the simulation community, it is necessary to conduct an experiment. This test was to determine if any conclusions could be drawn about the relative performance of the GemStone system. A database system should show improvements in speed when queries are centralized and give average results when the queries are randomized.

The following experiment involved five different terrain databases . There are four queries which will be sent to the database and each query will be sent a specified number of times. The timings given are the real clock.

Experiment

Each database was sent four queries with both randomized and localized data points. The queries that were used were :

elevationAt: aPoint
elevationAt: aPoint **put**: anotherPoint
intersects: anObject
visibilityFrom: aPoint **to**: anotherPoint

Each query was sent to the same database which was located in the Smalltalk-80 image and in the GemStone system. When the queries were being processed in Smalltalk-80, no activity was being conducted in GemStone, and the reverse was also true.

Each query was sent 1, 5, 10, and 50 times to each database. For each query sent, the distribution of the data points was either randomized or localized. Randomized points, in the case of the elevation and visibility queries, fell anywhere inside the legal boundary of the terrain. The randomized areas which were used in the **intersects**: anObject query came from the database itself. Two objects from the terrain part-whole hierarchy were chosen at random. The localized points were statistically close together. One seed point was picked at random. This along with a random range gave an enclosure which was random in nature. All points that are localized fall within that area.

Description of the Experimental Databases

There were five databases created and used in this experiment. Each database was created from the same raw data files but the amounts of terrain coverage by each database differs. Each database is also classified by the form of hierarchy its takes. For example, there are flat, balanced and skewed part-whole hierarchies. In addition, some of the databases contain terrain objects such as houses, trees, treelines, roadways and lakes. The following is a list of the five databases :

Database Number	Size Patches	Balanced/Flat/ Skewed	Objects or No objects
1	3x3	Balanced	Objects
2	3x3	Flat	No objects
3	2x2	Flat	No objects
4	3x3	Skewed	Objects
5	1x1	Flat	No objects

Results

The table which is located at the end of this chapter, Table 1, shows the numerical results which were obtained from conducting the experiment. All times are recorded in seconds. The queries are labeled at the top of the page while the number of repetitions are located at the side. Each database is marked with a number which corresponds to the list above.

There are three major conclusions that were discovered about the OOTDB from the result of conducting this experiment. The first is that the terrain database functions correctly. Both the Smalltalk-80 and GemStone queries returned the same results. The second observation is that the GemStone system is slow. It does not look like GemStone will be able to be used in its present configuration in a real-time simulation. Finally this experiment shows that more experiments need to be done. There is some obvious correlation of the data from the GemStone and Smalltalk-80 queries. But because GemStone is so slow, it is almost unnecessary to make any formal evaluation of its performance.

Table 1. Experimental Results

One of the inconstancies found was in the difference in times between the random and localized points. The localized queries should have taken less time to execute than their random counterparts. The data presented in the table shows that there are many times where this was not true. The reason these times differ from the expected pattern is unknown.

Possible Reasons for Poor Performance

There are a few possible reasons why such poor results were obtained from the GemStone system. First, the configuration of the software on the workstation which houses the GemStone server may be limiting its performance in some way. This particular computer is part of a network and one of its functions is to be an administrator. It has some resources which other workstations use.

Second, this database uses three dimensional data and all of the computation uses floating point arithmetic. It is not known if the GemStone server uses the underlying floating point processor, or must process all floating point numbers in software.

Additional Research Needed

This section is devoted to exploring additional work which needs to be done to better understand the usefulness of the OOTDB. Performance analysis, real-time implementations and future directions for the OOTDB will be discussed.

Performance Analysis

One important aspect of this project was that the design of the database and the spatial management system would be computationally efficient. The results of the experiment presented in chapter 6 did not cover any analytical study which would have determined order statistics for database algorithms. It only found that the use of GemStone as a real-time object oriented database system is not possible. An analytical study as well as experiments to verify the theoretical orders are needed.

Real-time Implementation

It is evident that the Smalltalk-80 and GemStone implementation presented in this report would not be suitable for a real-time simulation because the query processing is not fast enough. To be able to achieve real-time performance, a rehosting of the database in the C++ language may be necessary. There are some disadvantages is using C++ as a basis for the terrain database. C++ does not support a dynamic class hierarchy, which prevents any reorganization of the base class structure while the database is active. A change to a class must be done off-line and the database source code must be recompiled. An advantage to using C++ would obviously be the speed.

Another improvement which could be made to the terrain database is an analysis of the innermost loops and traversals of the database data structures. There may be some places where improvements can be made to the flow of code without changing its functionality. Once a final version of the database implementation is in place, it would then be interesting to look at the possibility of designing a database machine. This would be a hardware implementation of the terrain database, and it would be done to improve performance.

Future Directions For OOTDB

One important area of discussion that was brought out in this report was the use of object oriented designs and databases. The future of these technologies has impacts on the next generations of the OOTDB. But what is on the horizon for these products?

Distributed Processing

The main consensus is that object oriented research will be investigating distributed processing. It was evident at the OOPSLA'90 conference that concurrent and parallel implementations of object oriented products would be the next generation of research topics. Nine out of thirty two papers and panels were devoted to either concurrent objects or parallel object oriented programming [OOPSLA 90]. It is obviously an area of active research and will be so in the future.

To continue with the topic of distributed object oriented programming is the notion of a distributed OOTDB. It would be interesting to investigate how the terrain database can "live" on separate nodes while participating in one simulation. The class and part-whole hierarchies may be located on different nodes and the interaction of the simulation players on those nodes will determine where the different parts of the database will live. This line of thinking is in concert with the distributed and parallel research in object oriented programming above. Not only does this have enormous implications in the networking of simulators, but the implementation of a distributed terrain database may mean that thousands of players may be able to participate in a single virtual world.

There are many considerations when looking at distributed databases. One of the main questions asked is how to distribute the terrain data throughout the nodes of the simulation. Some objects may need to reside on more than one node. If a part of the terrain database needs to be used on one more than one node at a time, there is a question as to which of those places would actually get the terrain object, and which would have ghosts.

Another question to be answered would be how to establish communication between objects on different nodes. This raises questions in message route planning and the configuration of an underlying physical network.

A final area of investigation would be in the area of a distributed terrain class hierarchy. Would the entire class hierarchy be replicated on each node, or would some encapsulated version of the class hierarchy be distributed? These questions, as well as many that were not mentioned here will be the focus of work on the OOTDB in the future.

Effectiveness as a Training Simulation

Also worth mentioning are possible human experiments using the OOTDB. One of the important factors of using a real-time simulation is its ability to effectively train. Many times human factor experiments are used to determine how well subjects perform while using new equipment and software. This type of analysis is needed to determine how effective the OOTDB is in a simulation environment.

Physical Modeling

Another important aspect of this work is the connection of the database with other simulator components. There needs to be research in the areas of physical modeling, soil dynamics and dynamic terrain. These aspects of simulation along with a terrain database will improve the usefulness of a simulator. Physical and constraint modeling will give the database symbolic resolution of constraints which may be placed on terrain objects. Soil dynamics will determine exactly how the terrain will react to its environment. Dynamic terrain will lead to algorithms and data structures which will better represent the terrain as it changes. All of the simulator activities mentioned above should be supported by the OOTDB.

Bibliography

[Antony 90] Antony, R. "A Hybrid Spatial / Object Oriented DBMS to Support Automated Spatial, Hierarchical and Temporal Reasoning," Draft: Advances in Spatial Reasoning. Ablex Press, NJ, 1990.

[Antony 88] Antony, R. "Representation Issues in the Design of a Spatial Database Management System," United States Army Symposium on Artificial Intelligence for Exploration of the Battlefield Environment, El Paso, TX, November, 1988.

[Blau 90] Blau, B., "Object Oriented Terrain Databases for Visual Simulators," Masters Thesis, Department of Computer Science, University of Central Florida, Orlando.

[Johnson 88] Johnson, R. E., Foote, B. "Designing Reusable Classes," Journal of Object Oriented Programming, pp. 22-35, June/July 1988.

[OOPSLA 90] Conference on Object Oriented Programming: Systems, Languages and Applications and European Conference on Object-Oriented Programming, Ottawa, Canada, October 1990

Appendix L:

Distributed Databases for Dynamic Terrain

Dynamic Terrain Project
1990-91

3DBS Project
(Distributed Dynamic Database Based Simulation)

Project Name: 3dbs

Project Number: VSL 91.8

Author: Xin Li, Chuck Campbell

Date: Jan. 8, 1992

This document includes:

- Overview of the project
- Implementation scheme
- Analytic figures
- Tables of statistical data

Overview of the 3dbs project

I. Approaches

Centralized Database Centralized Simulator (CDCS)

A single simulator runs all vehicles on a centralized database. Display terminals hold a display database and receive user input. All terrain changes to a given patch are made to the centralized database and are forwarded to the display terminals which hold that patch in their display database.

Centralized Database Approach (CDA)

A centralized database is kept and accessed by each of the simulators. The simulators keep a local database on which they operate. When a vehicle modifies a patch of terrain, the changes are sent to the server, which updates the centralized database and forwards the changes to each of the other simulators, if any, which currently have a copy of the patch in *their terrain* database.

Centralized Database Approach with Counters (CDAwC)

This approach is very similar to the CDA approach described above. The server receives terrain changes for a patch and sends them to other simulators which currently have a copy of the patch in their local database as with the CDA. The difference is that the server keeps a counter for each patch which indicates how many simulators currently have a copy of that patch in their local database. As long as the counter for a patch is non-zero, the centralized database is not updated with terrain changes to that patch. As patches are released from simulator databases, the associated counter is decremented. When a counter becomes zero for a patch, it requests that the patch be sent back from the last simulator holding it so that the centralized database can be updated.

Distributed Database Approach (DDA)

The terrain database is divided into blocks of patches which are distributed among the workstations. The size of a block is arbitrary. Workstations consist of a simulator and a display. The display has a small database which holds the area around its vehicle. The display runs a dead-reckoning model of the vehicles it is displaying and receives changes in velocity and heading from the simulators. The display is always associated with the same vehicle. Vehicle simulation is performed on the simulator which holds the patch on which the vehicle is currently residing, so a vehicle is passed to a different simulator when it crosses a block boundary. This should cut down network traffic since terrain changes are always local to the simulator. The changes to a patch still need to be broadcast to any display databases which hold the patch.

II. Performance Analysis Centralized Database Approaches (using 4 vehicles)

The following are the test cases for the approaches which use a centralized database. They were developed to gather data for the various levels of interaction between vehicles during a simulation. Each of the cases have an expected result associated with them, which is stated with the case below. When more than one vehicle occupy the same patch, an update message is sent from each vehicle to every other vehicle on the patch, resulting in n^2 messages per iteration for n vehicles. Obviously a case which generates more messages per iteration is expected to take more time per iteration.

The statistics recorded for each of the cases include the number of messages sent and received per iteration, the number of bytes sent and received per iteration, the time taken for an iteration, and the number of iterations performed.

i. <1,1,1,1> All vehicles on different patches. This case is expected to generate 4 messages each iteration, one from each vehicle to the server. Since no patch is common to more than one vehicle, the server will issue no patch updates.

ii. <2,1,1> Two vehicles on the same patch, the other two vehicles on different patches. This case is expected to generate 6 messages (2*2+1+1) each iteration.

iii. <2,2> Two vehicles on the same patch, the other two vehicles on the same *different* patch. This case is expected to generate 8 messages (2*2+2*2) each iteration.

iv. <3,1> Three vehicles on the same patch, one vehicle on a different patch. This case is expected to generate 10 messages (3*3+1) each iteration.

v. <4,0> All vehicles on the same patch. This case is expected to generate 16 messages (4*4) each iteration.

III. Performance Analysis for the Distributed Database Approach (using 4 vehicles)

The performance analysis for the distributed database approach is more complex than that of the centralized database approaches. The expected number of messages for the DDA is variable. The occurs because a vehicle is passed between simulators, while its display remains static. If a vehicle's model is being run on a simulator which is part of the same workstation as that vehicle's display, updates to the display database can be made locally, without a network message. However, if the vehicle is being run on a different simulator, that simulator must send a network message to the vehicle display's database.

There is one exception. In the case where all four vehicles are being run on the same simulator, there will always be one update to the local display database, and 3 update messages sent to other display databases.

To illustrate the variability of network messages, consider the following example for the <3,1> case (see section II):

Vehicles: v1, v2, v3, v4

Workstations: (display:simulator)
d1:s1, d2:s2, d3:s3, d4:s4

Display Assignments: (vehicle:display)
v1:d1, v2:d2, v3:d3, v4:d4

expected #	vehicle/simulator relationship				description
of messages	s1	s2	s3	s4	of messages
2	v1,v2,v3			v4	s1->d2, s1->d3
3	v1,v2,v3		v4		s1->d2, s1->d3, s3->d4
4		v4		v1,v2,v3	s2->d4, s4->d1, s4->d2, s4->d3

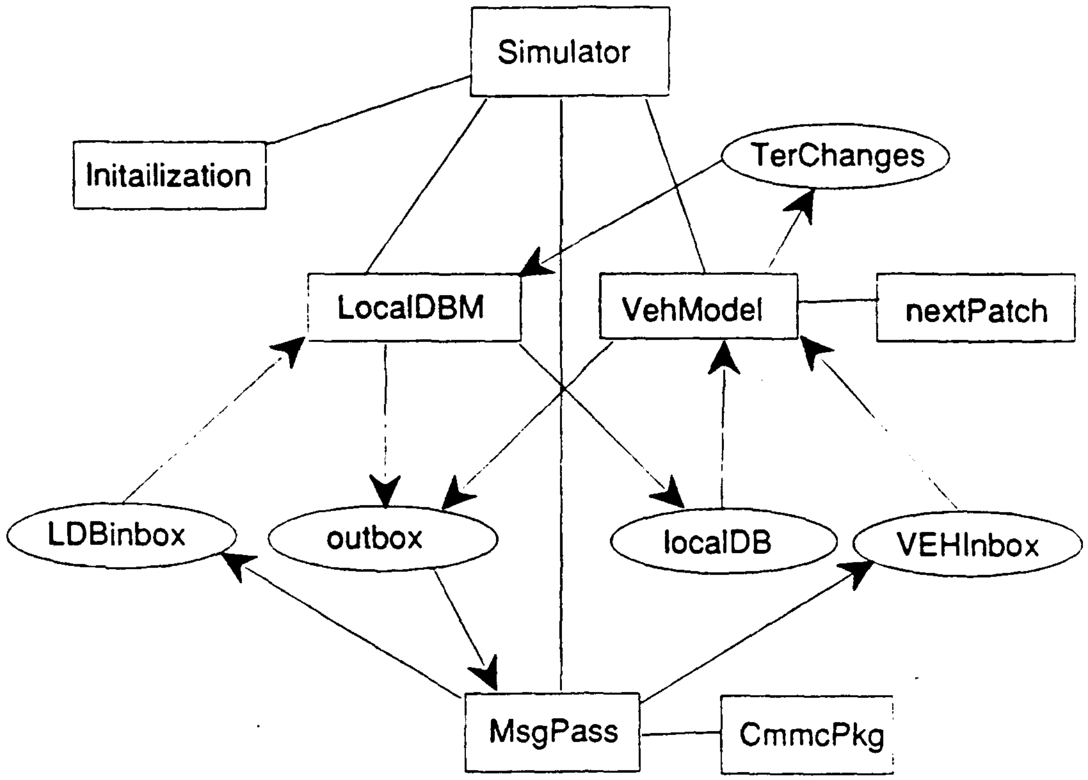
Of course, this is just one scenario for the <3,1> case, but it demonstrates the variability of the number of messages produced.

Using this approach, the number of expected messages for each case is as follows:

case	expected # of messages
<4,0>	<3>
<3,1>	<2,3,4>
<2,2>	<2,3,4>
<2,1,1>	<1,2,3,4>
<1,1,1,1>	<0,2,3,4>

Performance analysis will be performed on each instance of each case. These results will be reported in table form. For each case, the results will be averaged giving each instance equal weight for purposes of comparing the results of this method with the results of the centralized database approaches.

For more accurate results, one could determine the likelihood of each instance occuring during a simulation, and assign weights to the results reported in the tables. However, this is beyond the scope of this experiment.



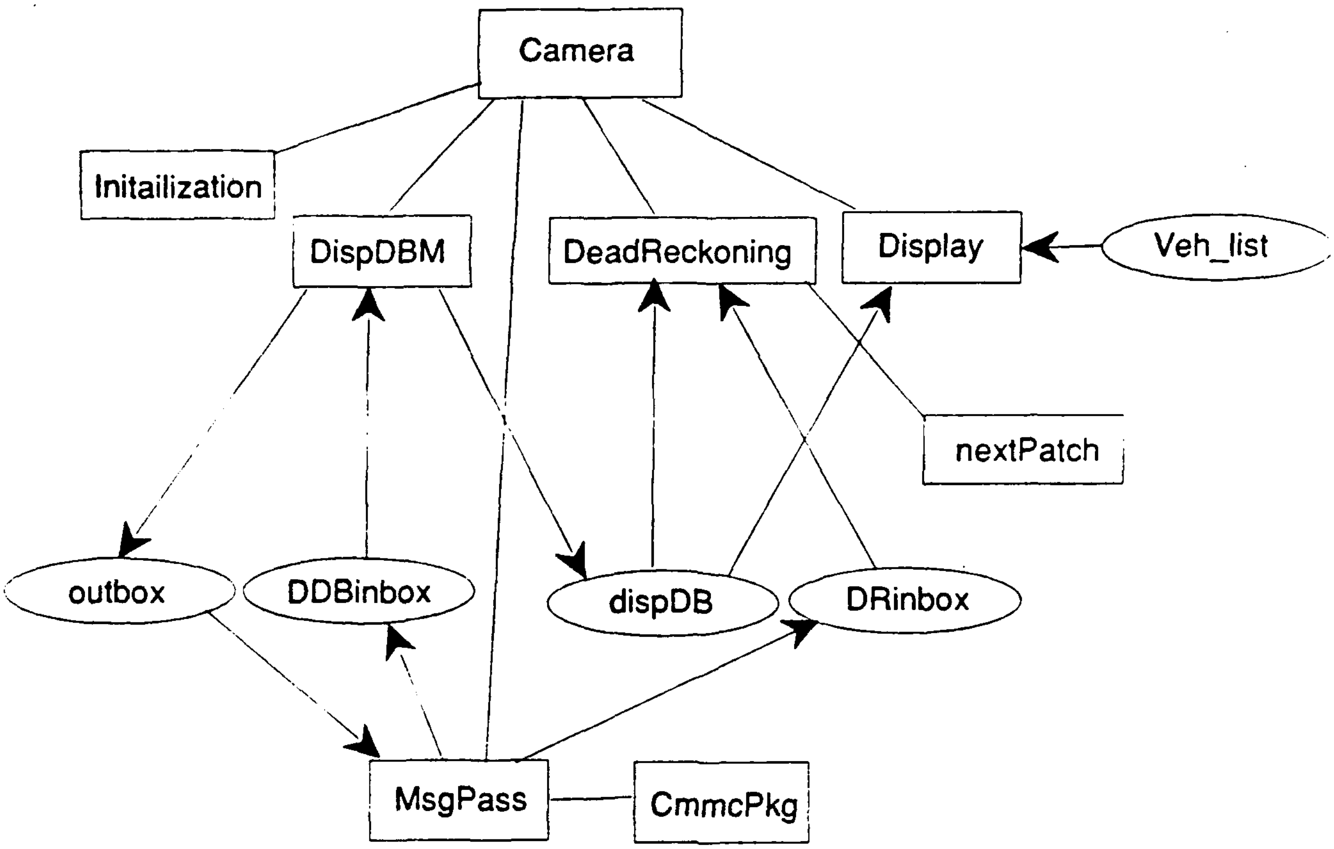
Functionality of Simulator:

1. LocalDBM:
   - i) receives patch requirements from inbox; Record patch numbers for update purpose;
   - ii) updates the local database by TerChanges and broadcast it to appropriate cameras;
   - iii) sends required patches to appropriate cameras.

2. **VehModel:**
   - i) receives keyboard & mouse input; calculates actual orientation and position of local vehicle;
   - ii) if local vehicle changes terrain, puts the modification in TerChanges (temporary storage);
   - iii) puts local vehicle update message into outbox;
   - iv) determines if local vehicle is crossing boundary, if so, pass message to corresponding simulator.

3. **MsgPass:** (see Camera)

4. **CmmcPkg:**
   sends messges either between processes or machines.



Functionality of Camera:

1. DispDBM:

i) checks current patches number and local vehicle to determines which new patches are needed. broadcasts those numbers by putting them into the outbox;

ii) checks inbox for terrain updates or new patch information; updates display database with new information.

2. DeadReckoning:

i) checks inbox for updated vehicle information;

ii) check display database and run dead-reckoning model for each active vehicle.

3. Display:

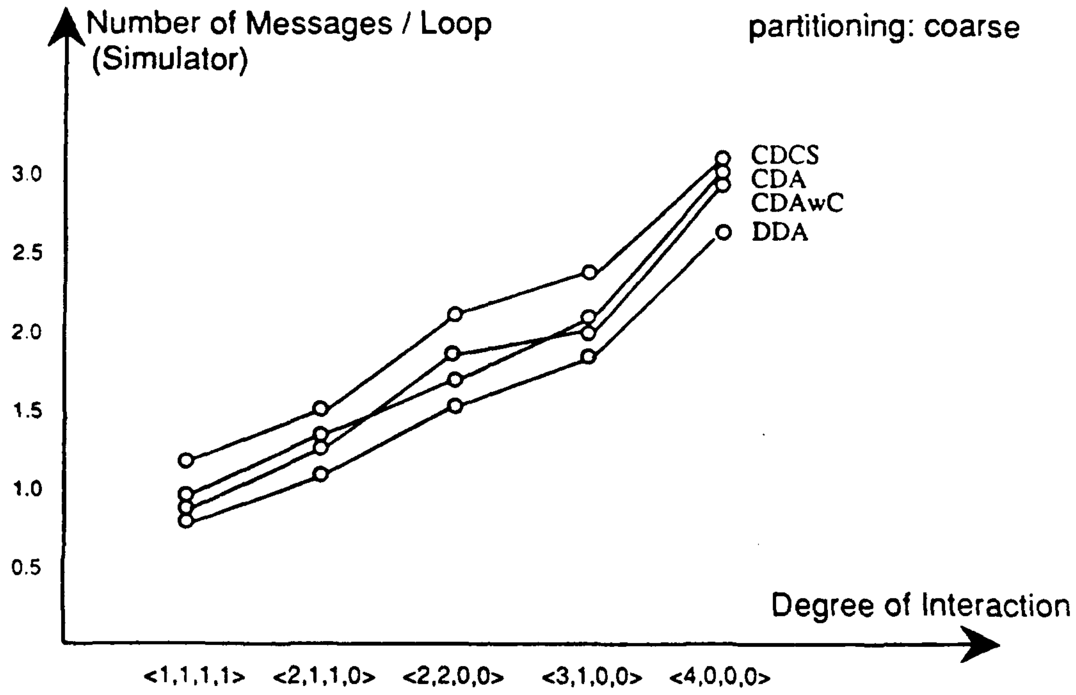
i) displays patches in camera diplay database and vehicles. (note: may provide 2 types of viewing: Overhead and window views. View depends on orientation of the local vehicle.)

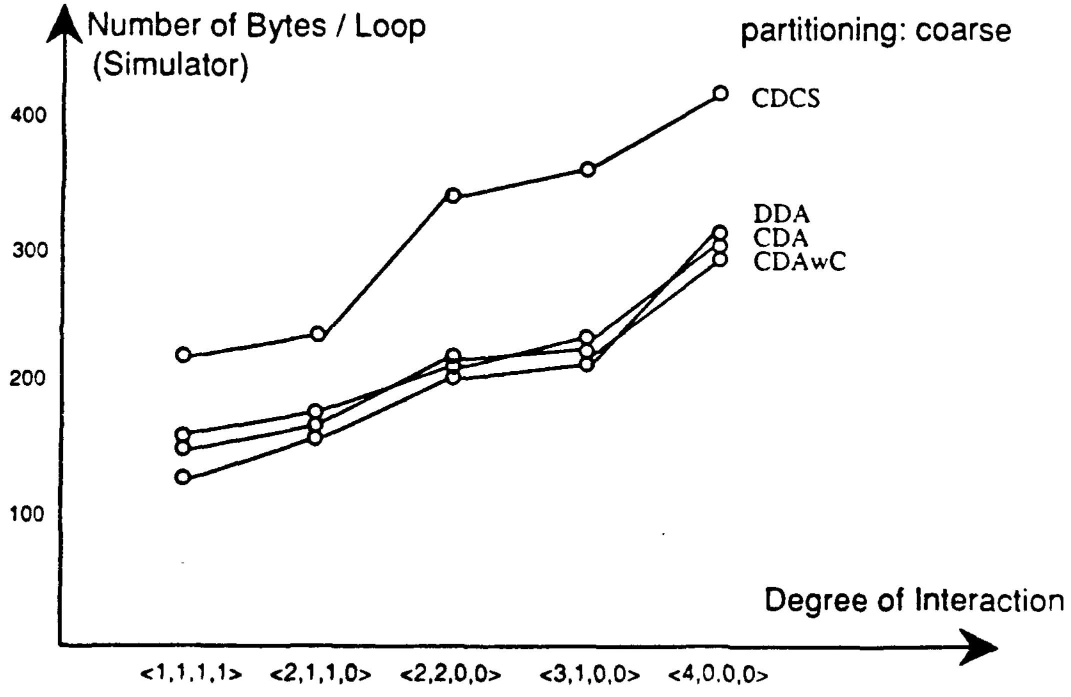
4. MsgPass:

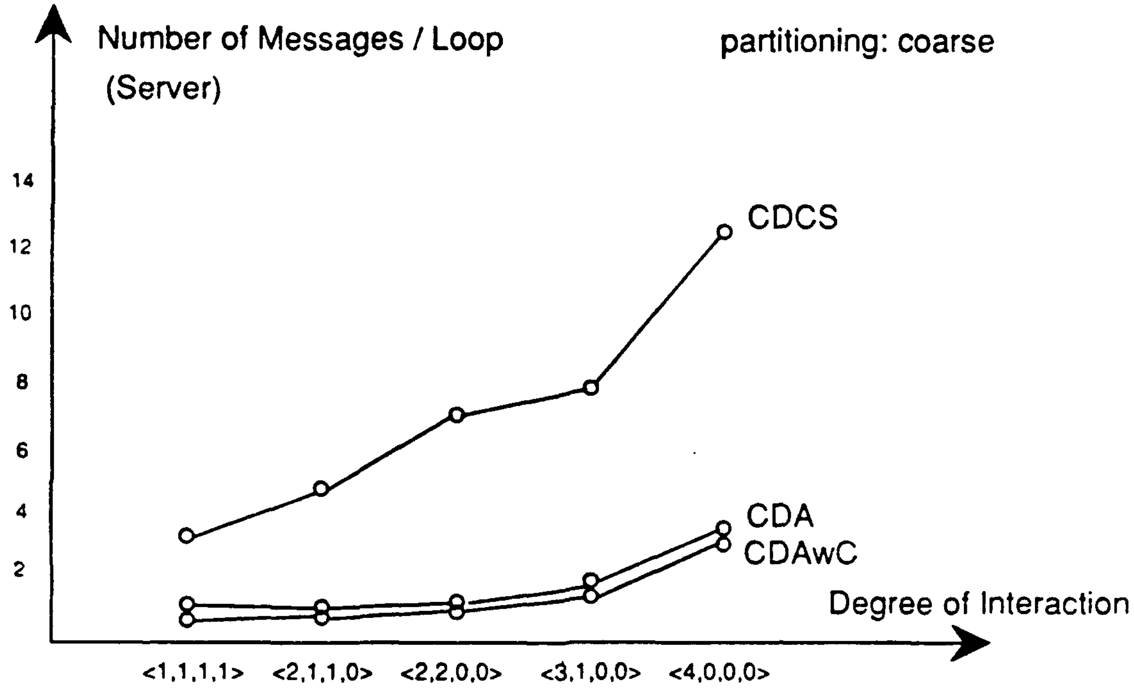
i) check outbox and pass the messages to this destination (broadcast or point to point).

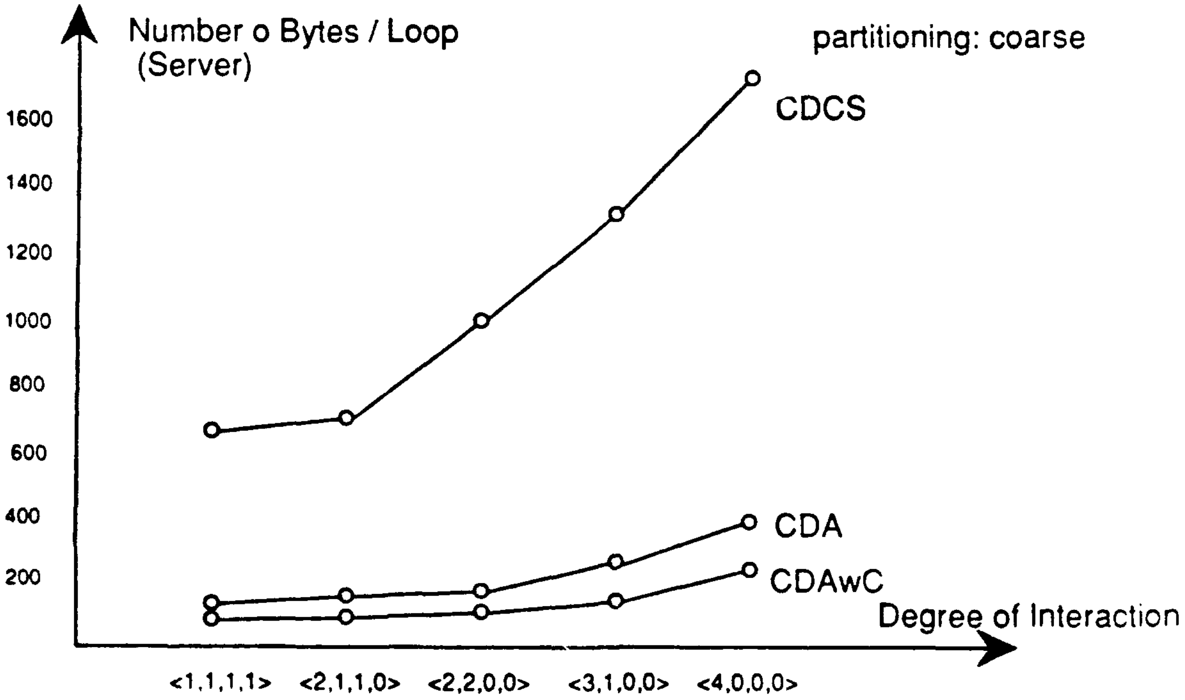
ii) receive messages and put into appropriate inbox.

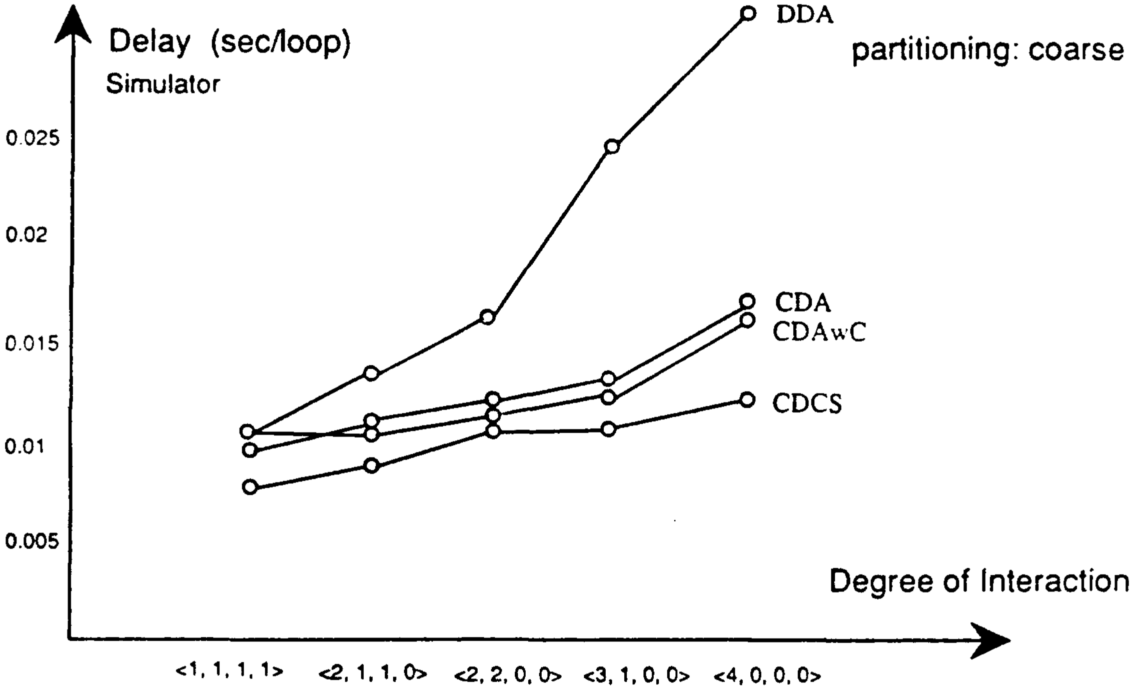
Coarse Partitioning

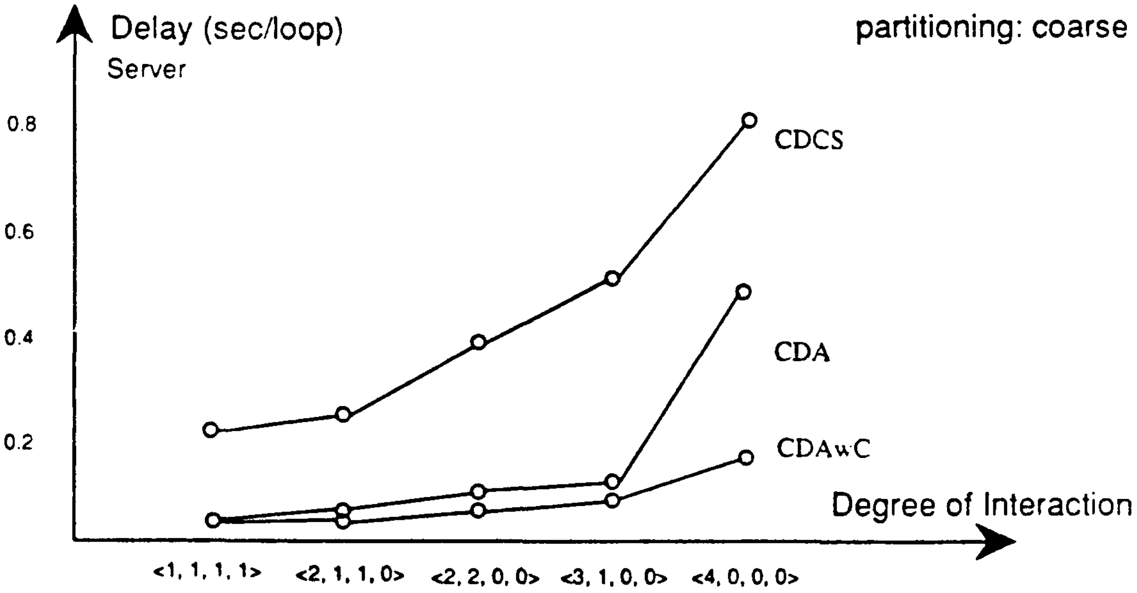

Number of Messages / Loop
(Simulator)
partitioning: coarse
3.0
2.5
2.0
1.5
1.0
0.5
CDCS
CDA
CDAwC
DDA
Degree of Interaction
<1,1,1,1>
<2,1,1,0>
<2,2,0,0>
<3,1,0,0>
<4,0,0,0>


Number of Bytes / Loop
(Simulator)
partitioning: coarse
400
300
200
100
CDCS
DDA
CDA
CDAwC
Degree of Interaction
<1,1,1,1>
<2,1,1,0>
<2,2,0,0>
<3,1,0,0>
<4,0,0,0>


Number of Messages / Loop
(Server)
partitioning: coarse
14
12
10
8
6
4
2
CDCS
CDA
CDAwC
Degree of Interaction
<1,1,1,1>
<2,1,1,0>
<2,2,0,0>
<3,1,0,0>
<4,0,0,0>


Number o Bytes / Loop
(Server)
partitioning: coarse
1600
1400
1200
1000
800
600
400
200
CDCS
CDA
CDAwC
Degree of Interaction
<1,1,1,1>
<2,1,1,0>
<2,2,0,0>
<3,1,0,0>
<4,0,0,0>





* Although the DDA has less network traffic than other aproaches, it has to do more work to manage its local database. The figure simply suggests that the DDA do not gain much performance by cutting off few messages in this simulation model. If computations for vehicle model and display database are more subtantial, or if long haul communications and slower baut rates are considered, the DDA could have better performance than other approaches.

system: simulator ***approach***: DDA ***Partitioning:*** coarse

	IN_msgs	IN_bytes	OUT_msgs	OUT_bytes	EXE_time
(4, 0, 0, 0)	2.48	320.6	2.43	308.3	0.034
(3, 1, 0, 0)	1.8	234.4	1.71	226.9	0.026
(2, 2, 0, 0)	1.52	233.4	1.43	226.3	0.017
(2, 1, 1, 0)	1.18	178.3	1.09	171.5	0.014
(1, 1, 1, 1)	0.83	138.1	0.75	132.2	0.011

system: simulator ***approach***: CDA ***partitioning***: coarse

	IN_msgs	IN_bytes	OUT_msgs	OUT_bytes	EXE_time
(4, 0, 0, 0)	2.273	244.2	0.773	58.83	0.018
(3, 1, 0, 0)	1.258	174.5	0.823	61.2	0.014
(2, 2, 0, 0)	0.895	154.8	0.832	63.1	0.013
(2, 1, 1, 0)	0.531	123.3	0.821	63.2	0.011
(1, 1, 1, 1)	0.141	94.3	0.835	63.25	0.01

system: server ***approach***: CDA ***partitioning***: coarse

	IN_msgs	IN_bytes	OUT_msgs	OUT_bytes	EXE_time
(4, 0, 0, 0)	0.992	74.44	2.829	304.8	0.4713
(3, 1, 0, 0)	0.726	54.145	0.807	152.4	0.118
(2, 2, 0, 0)	0.689	53.26	0.691	126.0	0.0905
(2, 1, 1, 0)	0.673	49.34	0.36	94.63	0.071
(1, 1, 1, 1)	0.664	50.88	0.051	71.53	0.052

system: simulator ***approach***: CDAwC ***partitioning***: coarse

	IN_msgs	IN_bytes	OUT_msgs	OUT_bytes	EXE_time
(4, 0, 0, 0)	2.21	239.7	0.783	56.8	0.017
(3, 1, 0, 0)	1.2	162.8	0.785	58.3	0.013
(2, 2, 0, 0)	1.06	162.3	0.793	58.6	0.014
(2, 1, 1, 0)	0.51	118.8	0.808	60.3	0.01
(1, 1, 1, 1)	0.14	86.8	0.783	58.1	0.012

system: server ***approach***: CDAwC ***partitioning***: coarse

	IN_msgs	IN_bytes	OUT_msgs	OUT_bytes	EXE_time
(4, 0, 0, 0)	0.559	41.09	1.593	171.8	0.172
(3, 1, 0, 0)	0.525	39.29	0.773	107.4	0.091
(2, 2, 0, 0)	0.506	37.83	0.516	94.0	0.067
(2, 1, 1, 0)	0.542	41.01	0.283	77.46	0.056
(1, 1, 1, 1)	0.642	48.1	0.057	67.45	0.053

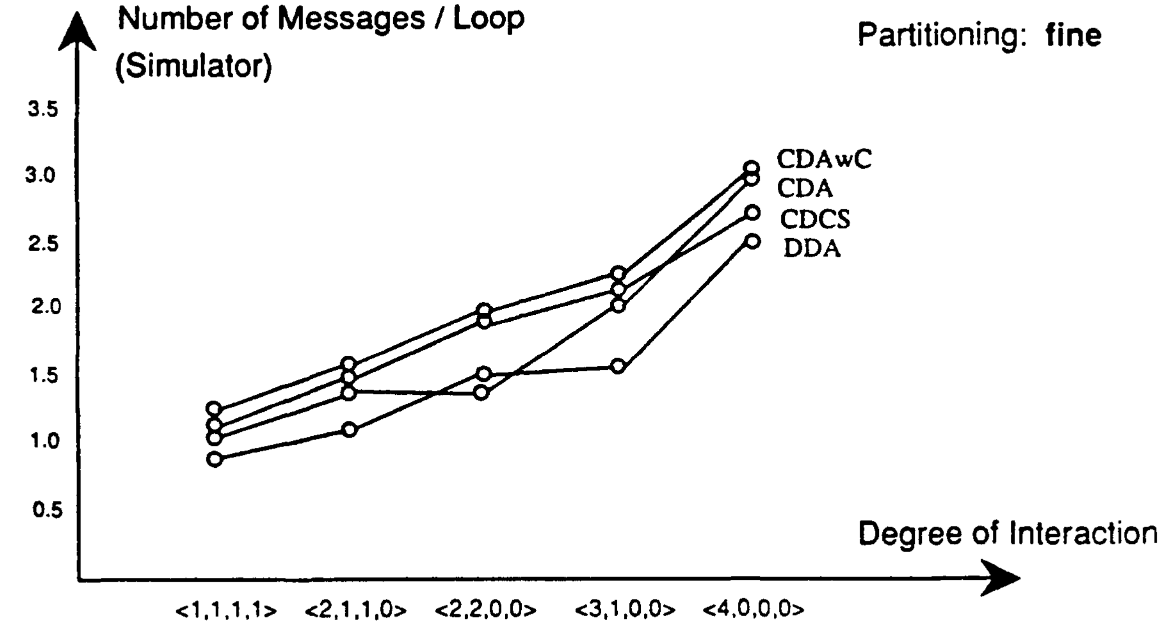
system: simulator ***approach***: CDCS ***partitioning***: coarse

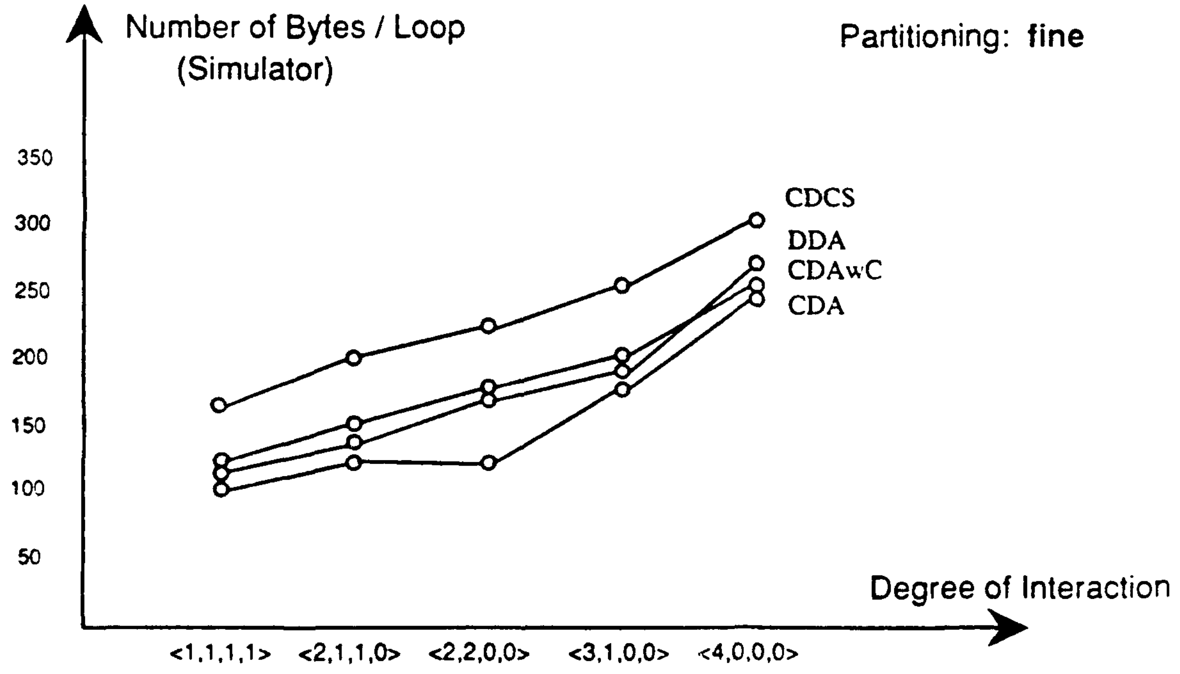
	IN_msgs	IN_bytes	OUT_msgs	OUT_bytes	EXE_time
(4, 0, 0, 0)	3.00	426.5	0.05	4.41	0.012
(3, 1, 0, 0)	2.31	364.2	0.05	4.37	0.011
(2, 2, 0, 0)	2.05	339.3	0.06	4.21	0.011
(2, 1, 1, 0)	1.43	227.3	0.08	5.01	0.009
(1, 1, 1, 1)	1.13	219.1	0.07	4.67	0.008

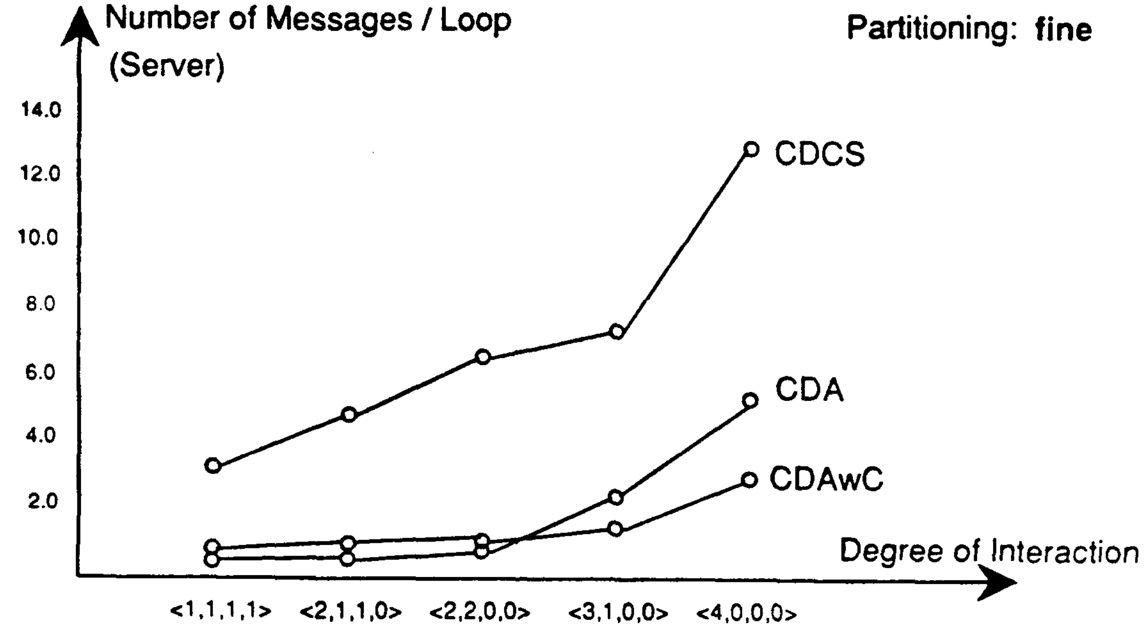
system: server ***approach***: CDCS ***partitioning***: coarse

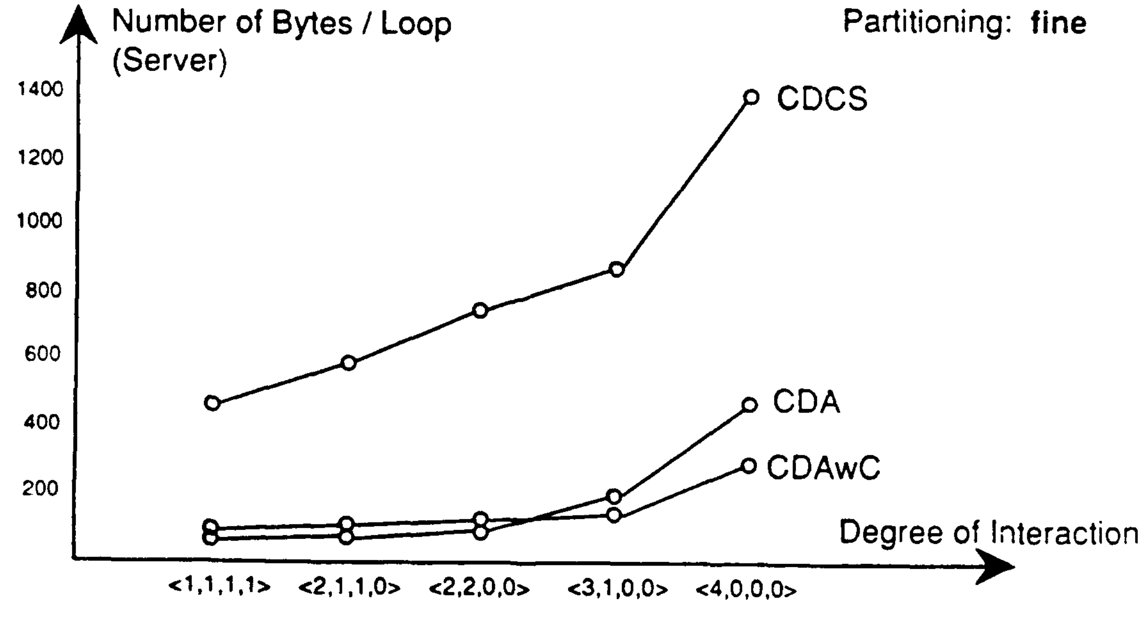
	IN_msgs	IN_bytes	OUT_msgs	OUT_bytes	EXE_time
(4, 0, 0, 0)	0.202	13.42	12.01	1725.7	0.818
(3, 1, 0, 0)	0.173	11.53	7.92	1288.6	0.493
(2, 2, 0, 0)	0.142	9.46	6.355	1018.7	0.377
(2, 1, 1, 0)	0.172	11.4	4.11	673.6	0.238
(1, 1, 1, 1)	0.152	10.1	3.07	637.9	0.213

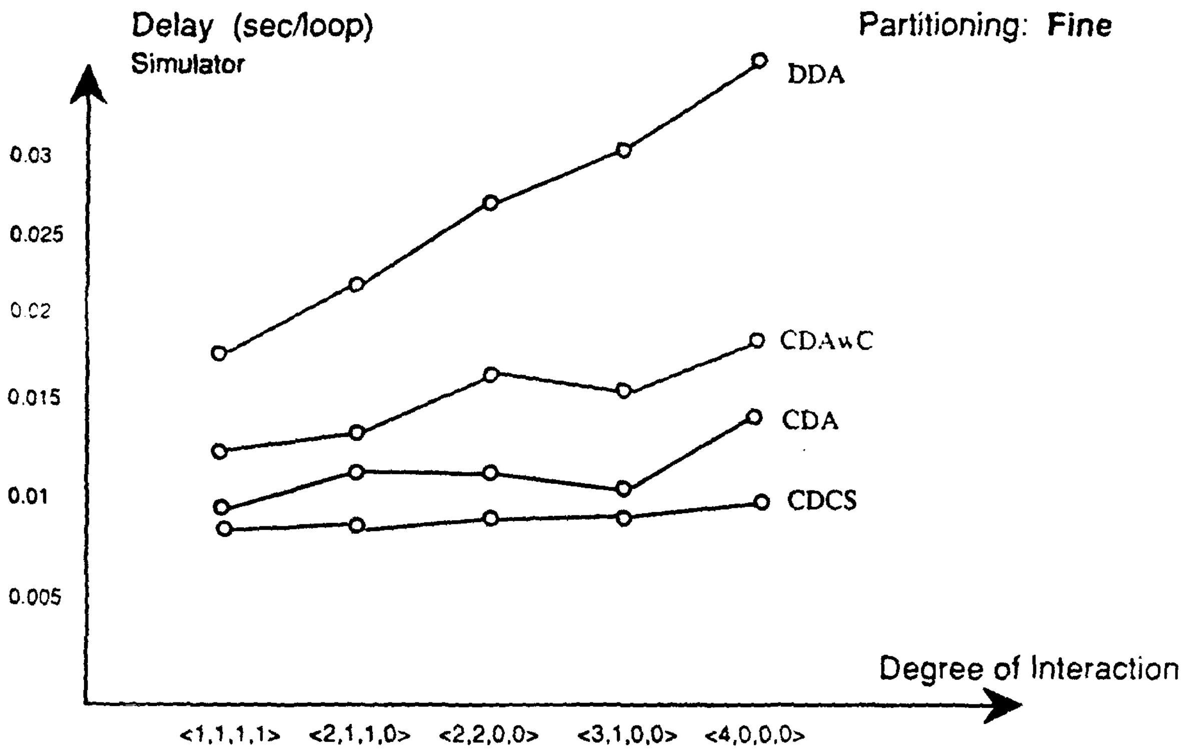
Fine Partitioning

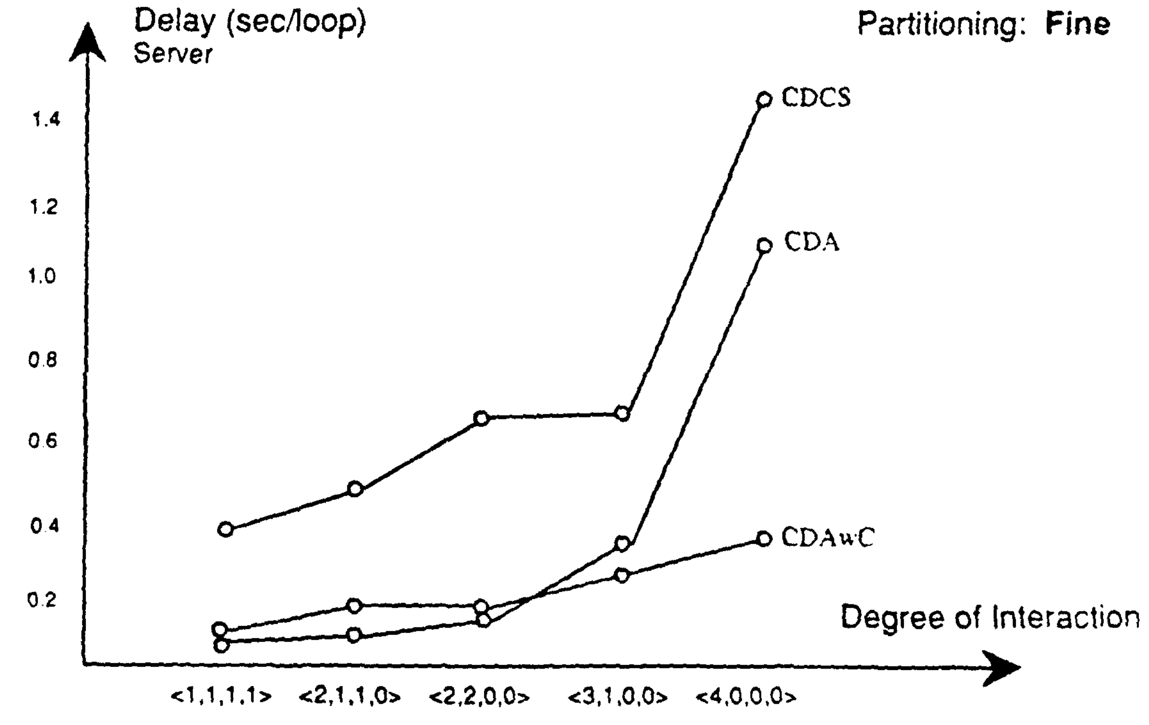

Number of Messages / Loop
(Simulator)
Partitioning: fine
3.5
3.0
2.5
2.0
1.5
1.0
0.5
CDAwC
CDA
CDCS
DDA
Degree of Interaction
<1,1,1,1>
<2,1,1,0>
<2,2,0,0>
<3,1,0,0>
<4,0,0,0>


Number of Bytes / Loop
(Simulator)
Partitioning: fine
350
300
250
200
150
100
50
CDCS
DDA
CDAwC
CDA
Degree of Interaction
<1,1,1,1>
<2,1,1,0>
<2,2,0,0>
<3,1,0,0>
<4,0,0,0>


Number of Messages / Loop
(Server)
Partitioning: fine
14.0
12.0
10.0
8.0
6.0
4.0
2.0
CDCS
CDA
CDAwC
Degree of Interaction
<1,1,1,1>
<2,1,1,0>
<2,2,0,0>
<3,1,0,0>
<4,0,0,0>


Number of Bytes / Loop
(Server)
Partitioning: fine
1400
1200
1000
800
600
400
200
CDCS
CDA
CDAwC
Degree of Interaction
<1,1,1,1>
<2,1,1,0>
<2,2,0,0>
<3,1,0,0>
<4,0,0,0>


Delay (sec/loop)
Simulator
Partitioning: Fine
0.03
0.025
0.02
0.015
0.01
0.005
DDA
CDAwC
CDA
CDCS
Degree of Interaction
<1,1,1,1>
<2,1,1,0>
<2,2,0,0>
<3,1,0,0>
<4,0,0,0>


Delay (sec/loop)
Server
Partitioning: Fine
1.4
1.2
1.0
0.8
0.6
0.4
0.2
CDCS
CDA
CDAwC
Degree of Interaction
<1,1,1,1>
<2,1,1,0>
<2,2,0,0>
<3,1,0,0>
<4,0,0,0>

system: simulator ***approach***: DDA ***partition:*** fine

	IN_msgs	IN_bytes	OUT_msgs	OUT_bytes	EXE_time
(4, 0, 0, 0)	2.62	280.5	2.4	262.3	0.036
(3, 1, 0, 0)	1.55	199.7	1.62	184.5	0.032
(2, 2, 0, 0)	1.54	191.3	1.38	177.3	0.026
(2, 1, 1, 0)	1.21	144.3	1.08	132.5	0.02
(1, 1, 1, 1)	0.96	118.5	0.81	105.2	0.016

system: simulator ***approach***: CDA ***partitioning:*** fine

	IN_msgs	IN_bytes	OUT_msgs	OUT_bytes	EXE_time
(4, 0, 0, 0)	2.2	192.7	0.84	58.8	0.0135
(3, 1, 0, 0)	1.2	121.3	0.84	59.0	0.0107
(2, 2, 0, 0)	0.48	70.9	0.80	54.9	0.0112
(2, 1, 1, 0)	0.58	73.2	0.82	56.2	0.0114
(1, 1, 1, 1)	0.22	54.3	0.80	55.9	0.0094

system: server ***approach***: CDA ***partitioning***: fine

	IN_msgs	IN_bytes	OUT_msgs	OUT_bytes	EXE_time
(4, 0, 0, 0)	1.6	114.9	4.15	357.6	1.079
(3, 1, 0, 0)	0.96	67.7	1.38	136.9	0.313
(2, 2, 0, 0)	0.62	43.1	0.51	59.8	0.11
(2, 1, 1, 0)	0.63	44.6	0.26	47.9	0.095
(1, 1, 1, 1)	0.66	46.9	0.099	35.6	0.073

system: simulator ***approach***: CDAwC ***partitioning:*** fine

	IN_msgs	IN_bytes	OUT_msgs	OUT_bytes	EXE_time
(4, 0, 0, 0)	2.22	195.5	0.84	58.2	0.0165
(3, 1, 0, 0)	1.49	141.7	0.85	63.5	0.0147
(2, 2, 0, 0)	1.16	119.8	0.84	62.7	0.016
(2, 1, 1, 0)	0.76	90.1	0.82	62.2	0.013
(1, 1, 1, 1)	0.41	66.6	0.86	66.8	0.0124

system: server ***approach***: CDAwC ***partitioning***: fine

	IN_msgs	IN_bytes	OUT_msgs	OUT_bytes	EXE_time
(4, 0, 0, 0)	0.78	52.3	2.12	198.2	0.315
(3, 1, 0, 0)	0.67	49.9	1.09	102.8	0.225
(2, 2, 0, 0)	0.6	44.9	0.76	77.1	0.164
(2, 1, 1, 0)	0.63	49.1	0.52	61.2	0.176
(1, 1, 1, 1)	0.66	51.6	0.23	42.4	0.0929

system: simuiator ***approach***: CDCS ***partitioning:*** fine

	IN_msgs	IN_bytes	OUT_msgs	OUT_bytes	EXE_time
(4, 0, 0, 0)	2.63	290.7	0.093	6.18	0.0098
(3, 1, 0, 0)	2.18	252.3	0.108	6.88	0.0093
(2, 2, 0, 0)	1.91	229.7	0.105	6.78	0.0089
(2, 1, 1, 0)	1.55	193.5	0.093	6.03	0.0081
(1, 1, 1, 1)	1.23	162.3	0.086	5.53	0.0091

system: server ***approach***: CDCS ***partitioning***: fine

	IN_msgs	IN_bytes	OUT_msgs	OUT_bytes	EXE_time
(4, 0, 0, 0)	0.41	26.3	12.65	1418.6	1.428
(3, 1, 0, 0)	0.31	20.2	7.45	885.9	0.645
(2, 2, 0, 0)	0.30	19.6	6.3	788.7	0.625
(2, 1, 1, 0)	0.27	17.3	4.69	601.9	0.435
(1, 1, 1, 1)	0.25	15.9	3.37	475.7	0.345

Appendix M:

Virtual Environment Realtime Networks

Dynamic Terrain Project
1990-91

Final Report on the Virtual Environment Realtime Network Project

B. Blau, C. E. Hughes, J. M. Moshell, L. Xin and J. Chen
Institute for Simulation and Training
University of Central Florida
Orlando, Florida

The Virtual Environment Realtime Network (VERN) was a series of experiments designed to study the feasibility of distributing a virtual environment across multiple graphics workstations. Using object oriented technology and the dead reckoning concept borrowed from SIMNET, a set of programs was developed to demonstrate our ideas. This has resulted in four versions of the software (available on Sun Sparcs and Silicon Graphics workstations) and two published papers in technical conferences [Appendix C, Moshell 91].

The main design goal of VERN was to look at object oriented technology to develop this virtual environment. Our first demonstration program [Appendix A] was developed in Smalltalk-80, a pure object oriented language. The concept of a Player and Ghost were developed by the staff and students at IST.VSL. Each real world object participating in the simulation is represented by a software object called a Player. The Player resides on the object's home machine. If human or external input is required by the Player, the data is read and processed on the Player's home machine. The main responsibility of the Player is to accurately maintain state information, read and process inputs, provide feedback usually in the form of real-time graphics, and inform the network of any significant state changes that deviate from the dead reckoning model.

In order to facilitate communication between Players residing on separate machines, each Player has an associated Ghost located on every machine involved in the simulation. Thus in an N Player simulation on M networked machines, each machine is guaranteed to have exactly N objects representing all players. Such a configuration allows Players to communicate locally with any other Player (represented by its Ghost). It is the responsibility of the Ghost either to respond directly to the message, or to forward it to the actual Player.

Ghosts are approximations of their associated Players. That is, the state of a Ghost is not always as precise (algorithmically) as the Players, but this approximation is adequate for visualization and dynamics. All Ghosts that are associated with a single Player are synchronized at any given instant in simulation time through the use of the system clock, message passing and dead reckoning. When the Player realizes that its Ghosts are going to be inaccurate, the Player then communicates the correct state information to all Ghosts.

This version of VERN was executable only in the Smalltalk-80 environment and no graphics were available. The next target platform was the Silicon Graphics workstations. This meant a move from Smalltalk-80 to C++. This move was accomplished and C++ is the current programming language. The main features of the last version of VERN include :

- Supports multiple objects across multiple machines
- Dynamic dead reckoning thresholds
- Automatic updates of Ghosts at predetermined intervals
- Support of TCP/IP sockets

There are still unanswered questions and problems yet to be solved. First, we did not design the software to run over long distances. This addition is possible with the current software and would not require a redesign. Additionally, there is no master control paradigm in VERN. This means that programs compiled using the VERN classes produce an executable program that can run on a workstation. The design of a user interface as a control device is important and would enhance this project.

Appendices

Appendix A : VSL Document 90.14 Dynamic Terrain Project:The Virtual Reality Testbed Smalltalk Prototype.

Appendix B : VSL Document 91.4 How To Implement Networked Simulation Players Using VRTB v1.05.

Appendix C : This paper is to be published at the 1992 Symposium on Interactive 3D Graphics in March of 1992.

References

Moshell, J. M., et. al, "Networked Virtual Environments for Simulation and Training," *1991 International Simulation Technology Conference,* Orlando, FL, Oct, 1991.

APPENDIX A

VSL Document 90.14

Dynamic Terrain Project:
The Virtual Reality Testbed Smalltalk Prototype

Charles E. Hughes, Brian Blau, Xin Li, J. Michael Moshell

Date written : 11/24/90
Date printed : 1/2/92
VSL Memo 90.14

Abstract

The **Virtual Reality Testbed (VRT)** protocol forms the software basis for an environment that will support experiments with a network of visual simulators operating in a pseudo-universe. This universe will contain animate as well as inanimate objects. Some might exhibit behaviors that are controlled by a human operator, while others may have fully or semi-automated behaviors. Objects that communicate with each other may exist on the same or separate computers. In fact, objects might even move from one computer to another during a simulation. Of paramount importance is that no object need know the location of another. All may assume that they have a direct link to every other object and that performance is not altered based on the actual physical locations of these objects.

This report outlines the concepts and describes a prototype for the high-level protocol of the VRT. Smalltalk code which implements this prototype is also included and analyzed. Details on how network connections are made, used and closed are not included here.

I. Players and Ghosts.

In what follows we will make use of the terms **Player** and **Ghost** . Each object participating in a simulation is represented by a software object called a Player which resides on the object's home computer. If human or other external input is required by an object, this input is read and processed by the Player. The Player is responsible for computing and maintaining precise state information of its object.

Since one Player might be on a completely different machine than another with which it needs to communicate, each Player will have many Ghosts, one per machine that contains an interacting Player. Ghosts are approximations to their associated Players. That is, the state of a Ghost is not always a precise reflection of the Player's state, but is a "good enough" approximation. All the Ghosts associated with a single Player are synchronized in the sense that, for any given instance in simulated time, all report the same state information. Players are responsible for discovering when their Ghosts are about to report poor approximations, and sending ***update*** messages to inform the Ghosts about their new correct states.

All communications between Players are mediated by Ghosts. ***Queries*** (requests for state information) are completely handled by Ghosts who merely report their own states. ***Commands*** (requests for state changes, e.g., "Follow Me" or "You're Dead") are sent to Ghosts and relayed to their associated Players.

The problem of keeping Ghosts and Players synchronized is one of the more challenging ones that is faced in the VRT. To solve this problem, we have divided each discrete interval of time (called a tick) into two phases (initiated by the messages ***tick1*** and ***tick2***.) The *tick1* phase is provided so that Players and Ghosts can absorb state change messages sent during the previous tick. Ghosts receive *update* messages sent by Players, and Players receive *command* messages sent by other Players and relayed by the appropriate Ghost resident on the sending Player's machine. In this phase, Ghosts are also responsible for computing an approximate new state. All the Ghosts of a given Player must use the same approximation algorithm. In keeping with SimNet terminology, we refer to these approximations as ***Dead Reckoning Algorithms***.

The *tick2* phase is when each Player computes its new state, checks it against the Ghost's presumed next state, and sends an *update* message if the Ghost's next approximation (by *deadReckoning*) is not "good enough". The measure of "goodness" is, of course, dependent on the class of object being modeled by this Player. *Commands* received during this phase, must be held in a deferred queue until the start of the next interval. During *tick2* Ghosts play a rather docile role, responding to *query* messages, relaying *commands* that are intended for their Players, and remembering any *updates* that arrive. To avoid timing problems, Ghosts may not use their updated states until the next time interval.

II. Getting a Simulation Started

To test out the concepts in our VRT, we developed a Smalltalk program. A simulation is started by sending a ***startup*** message to the **VRT** class. The *startup* process will create new a instance of **Random**, called **Rand**, and one of **VRTClock**, called **Clock**. It then forks a process running the Clock method ***tick*** and sends the message ***init*** to the class **Network**. At this point *startup* is complete, and the simulation progresses under control of the Clock object.

The random number object, Rand, is used in the simulation to create random length delays, thereby simulating network traffic and process contention. The Clock object controls the high-level flow of the simulation, sending clock ticks and receiving requests to add new players. Both of these objects are accessible to all other simulation objects via the VRT class variables Rand and Clock.

Upon receiving *init*, the Network class creates a number of **Nodes**, each of which represents a computer (node, machine) in the network, and each of which exists as a separate process awaiting packets over the simulated network. Each Node is also sent the message ***wannaPlay***. Upon receiving the message *wannaPlay*, a Node (machine) on the network returns a series of packets, each of which contains a **PlayerDescription** of a **Player** who resides on this Node and wishes to be part of the simulation. A PlayerDescription object specifies the name of the Player, its class (e.g., Tank, Fish, Tree) and the machine id of this node.

As mentioned above, the Clock object has responsibility for accepting new Players and sending out clock ticks. The Clock inspects its **playerPackets** queue at the start of each time cycle to see if any new PlayerDescriptions have been recently sent. If any have been, Clock broadcasts a ***make:*** message which contains all the PlayerDescriptions. Each node then creates its own Players and one Ghost for each Player participating in the scenario.

Once each Node completes its make activity, it sends a ***done*** message back to the Clock, using the queue **donePackets**. When all Nodes have reported back, the Clock sends a ***startup*** message to the Nodes, who then distribute the startup to all resident Players. The default action for *startup* by a Player is to send an ***update:*** message to its Ghost. Since Ghosts always relay updates (through the ***remoteUpdate:*** message) to their associated Ghosts on all other Nodes, this will force every Ghost of a given Player to have the same state at the start of the *tick2* phase of the this time interval. Each Node must again send a *done* message to the Clock when the startup process completes.

After starting up all new Players, if there are any, the Clock sends the *tick1* and then the *tick2* message to all Nodes. A new tick is sent only after the previous tick has had its full effect. This synchronization is achieved by having the Clock wait until all Nodes have sent a *done* message, indicating that their Players and Ghosts are done.

This cycling between starting new Players, sending *tick1* and then *tick2* goes on until we somehow decide to end the simulation. At present, this is done after some fixed passage of simulated time.

III. Synchronization

The protocol discussed above, and elaborated below, seeks to avoid race conditions in which the order of message handling could affect the results of a simulation. This two phase ticking mechanism guarantees that *queries* and *commands* are handled in a consistent way. It does not, however, address how the simulation Clock knows when a phase has completed so it can send the next tick. A factoring of this problem is achieved by requiring each Node to send a *done* message when its Players and Ghosts have all completed. Looking back at the *startup* , *tick1* and *tick2* message, we see that we have already assumed that this is being done in those cases. In fact, the problem seems to be easy to solve if we assume that the Players and Ghosts on a given Node run in a single process that includes the Node itself. Under this assumption, the Node can call its objects in some predetermined, but unimportant order. The Node will send a tick message to an object and, when this object call completes, send the tick to the next object, until all have been served.

The purely sequential solution posed above works when all communication is intra-node, but fails when inter-node communication occurs. In our protocol a Ghost often communicates with its counterparts (Ghosts and Player) on other Nodes. This occurs when an *update* message is relayed to other Ghosts and when a *command* message is relayed to a Player. The current solution to this problem requires an ***acknowledge*** message to be sent by the receiver as soon as it completes handling the relayed message. This acknowledgement is not to the sending object, but is rather to the sending Node. The Node then determines that all inter-node messages are processed whenever it receives as many acknowledgement as messages it has sent.

IV. Vocabulary

The previous section introduces a fair amount of vocabulary that it will be useful to summarize before we look at a coded prototype.

Simulation Objects:

VRT Class: Starts a simulation by sending creating and forking the Clock, and sending the *init* message to the Network class.

Node: Receives all messages and all acknowledgements from other Nodes. This includes the Clock ticks and the make and startup messages. The Node represents a machine in a network. Its process includes the Players, Ghosts and PlayerDescriptions present on this machine. Nodes are instances of a class Network which includes several class methods to help manage the Nodes.

Clock: Acquires and makes known the presenece of new Players. It also sends tick messages to the Nodes telling them to advance time (*tick1*) or carry out the second phase of the current time (*tick2*).

Player: Encapsulates the "real" state and the methods needed to compute each new state for an actual participant in the simulation. Each Player resides on exactly one machine (Node).

PlayerDescription: Describes a Player by giving its name, home machine (Node) and its class. Each participating Node keeps a PlayerDescription of all Players that are in the simulation but do not reside on this Node. During the simulation, the PlayerDescription objects are used to relay messages to the "real" Player.

Ghost: Encapsulates an approximation to the state of its associated Player. Every Node contains a Ghost for all the Players in a simulation. Ghosts are responsible for answering Queries and relaying Commands to their Players. The VRT protocol is designed to guarantee that all the Ghosts for a given Player respond with precisely the same state information whenever a Query is issued. This provides a consistent behavior independent of the physical placement of the Player. For simplicity we will hereafter refer to a Ghost on the same Node as its Player as a resident Ghost. Others are called non-resident.

Messages:

wannaPlay. This is performed by each Node with the intent of to sending to the Clock PlayerDescriptions of all Players on this machine that will participate in th current simulation. Nodes return these PlayerDescriptions by a packet protocol implemented through SharedQueues in Smalltalk.

make:. This is broadcast from the Clock to all Nodes reporting to them the collection of all new PlayerDescriptions who have just entered the current simulation. A Node creates a Player object for each Player residing on that Node, a PlayerDescription object for each non-resident Player, and a Ghost for every Player whether resident or not.

startup. This is broadcast by the Clock to all Nodes telling them to start all their Players. The Node distributes the *startup* message to each of its resident Players. Each Player is expected to compute its initial state and send an *update:* message to its resident Ghost. A Ghost should relay the *update:* to all other Nodes so that each one instructs its corresponding Ghost to have the correct initial state. This message is also sent when a new Player arrives after the simulation has started. *Startup* follows *make* and precedes *tick1* for the current interval.

tick1. This is broadcast by the Clock to all Nodes telling them that a new time interval is starting. Each Node distributes the *tick1* message to all its Players and Ghosts. During the *tick1* phase, each Player processes *commands* from the previous time interval. Each Ghost sets its new state based on either a *deadReckoning* algorithm or the state from an *update* message received during the previous time interval. It then approximates its NextState, using *deadReckoning*. The Nodes are responsible for determining when their Players and

Ghosts have completed all activities in this phase. At completion, each Node sends a packet with the message *done*.

deadReckoning. This is used by Ghosts to compute their next state approximations.

tick2. This is also broadcast by the Clock. It tells all Nodes that the second phase of the current interval is starting. Each Node distributes this message to its Players and Ghosts. Each Player computes its next state and then sees how closely this matches the state approximated through *deadReckoning*. If the approximation is not good enough, the Player sends an *update* message to its Ghost.

update:. This is sent by a Player to its resident Ghost. The *update* message passes the Player's new state which is copied by the Ghost and then relayed to the non-resident Ghosts for this Player.

remoteUpdate:. This is sent from a resident Ghost to a non-resident Ghost, using Node-to-Node communication. It serves the same purpose as an *update*, but does not result in further relaying.

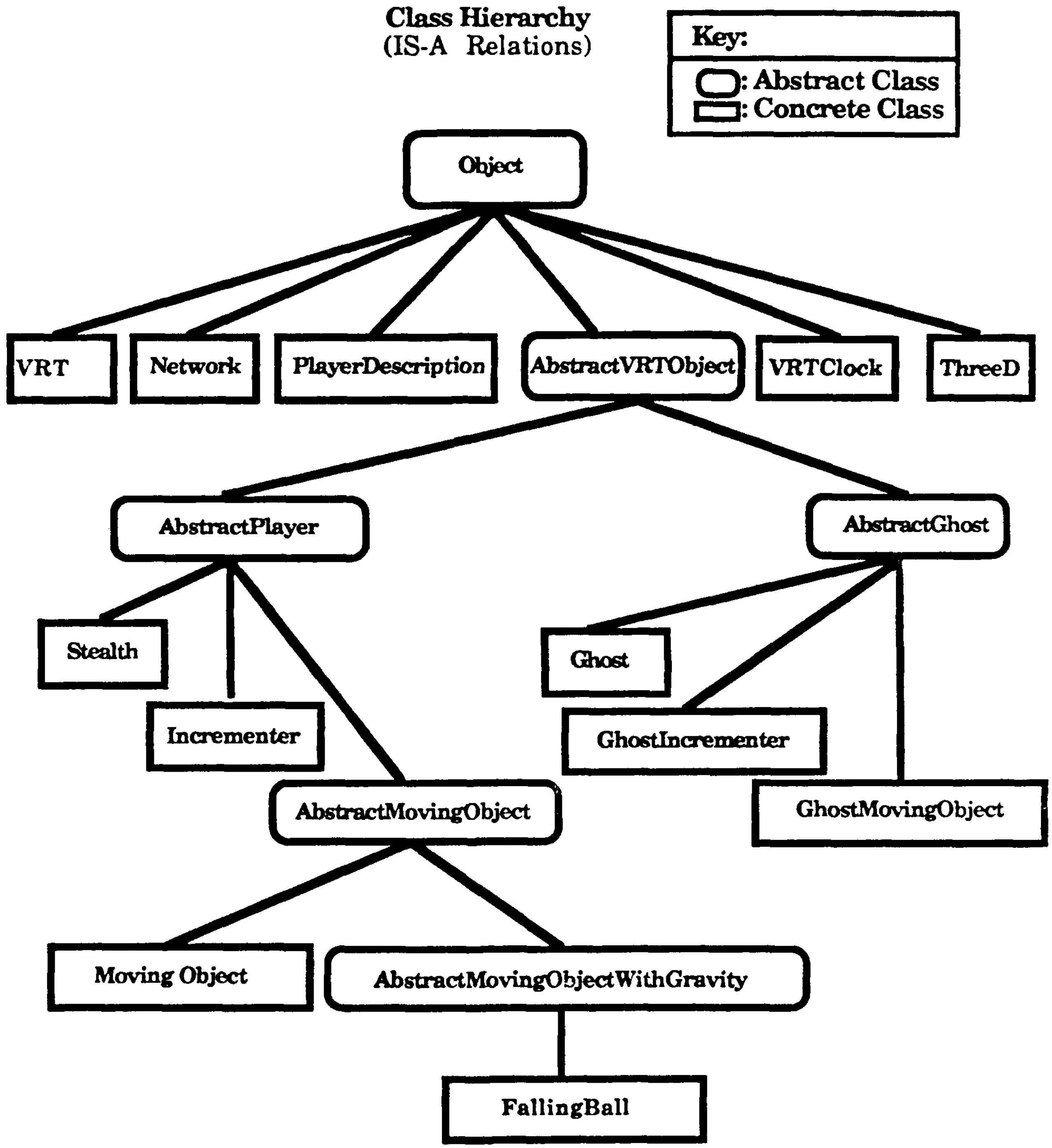
query. This is sent from a Player to a Ghost. The Ghost returns its current state which is an approximation to that of its Player. Queries can be sent in only the *tick2* phase.

command. This is sent from a Player to another Player, using the receiving Player's Ghost as an intermediary. Commands could include messages such as "FollowMe". The actual interpretation of any command is determined by the receiving Player's protocol. In any case, *commands* are sent in only the *tick2* phase and processed in only the *tick1* phase.

acknowledge. This is sent by a receiving Node back to some message sending Node. It should be sent only after the sending Node's message has been fully handled. The most common situation for an *acknowledge* is when a non-resident Ghost processes a *remoteUpdate*.

done. This is sent by a Node to the Clock using the donePackets queue. This is used to report completion of handling the *make* or *startup* message and to report that the current tick phase has been completed.

V. Hierarchy of Smalltalk Prototype



APPENDIX B

VSL Document 91.4

How To Implement Networked Simulation Players Using VRTB v1.05
VSL Memo 91.4

Date written : 4/19/91
Date printed : 1/2/92

Brian Blau
Visual System Laboratory
Institute For Simulation and Training
Orlando, Florida

Caveat

It should be noted that this work is in a design and review stage. The validity of the algorithms and design have yet to be proven effective and correct. The intent is to critique this work as well as finalize the ideas. Hopefully, the efforts of a large group will make the deisgn of the VRTB as robust as possible.

Introduction

The **Virtual Reality Testbed** (VRTB) protocol forms the software basis for an environment that will support experiments with a network of visual simulators operating in a pseudo-universe. This universe will contain animate as well as inanimate objects. Some might exhibit behaviors that are controlled by a human operator, while others may have fully or semi-automated behaviors. Objects that communicate with each other may exist on the same or separate computers. In fact, objects might even move from one computer to another during a simulation. Of paramount importance is that no object need know the location of another. All may assume that they have a direct link to every other object and that performance is not altered based on the actual physical locations of these objects.

Implementation of specific simulation objects using VRTB consists of writing C++ programs which follow the VRTB protocol. This document describes the specific implementation details without giving information about how the VRTB internals work. Section VRTB Overview gives a brief description of the internal workings, but a detailed explanation is beyond the scope of this document.

It is assumed that implementers have a working knowledge of general programming techniques, C and C++ languages and object oriented principles. The following terms should be known; class, superclass, subclass, instance variable, method, casting, inheritance, polymorphism, class hierarchy, abstract and concrete class.

VRTB Overview

Each object participating in a simulation is represented by a software object called a **Player** which resides on the object's home computer. If human or other external input is required by an object, this input is read and processed by the Player. The Player is responsible for computing and maintaining precise state information of its object.

Since one Player might be on a completely different machine than another with which it needs to communicate, each Player will have many **Ghosts**, one per machine that contains an interacting Player. Ghosts are approximations to their associated Players. That is, the state of a Ghost is not always a precise reflection of the Player's state, but is a "good enough" approximation. All the Ghosts associated with a single Player are synchronized in the sense that, for any given instance in simulated time, all report the same state information. Players are responsible for discovering when their Ghosts are about to report poor approximations, and updating then accordingly. The notion that a Ghost can compute limited state information is called **dead reckoning** and is an important aspect of the VRTB.

Main VRTB Components

The implementation of the VRTB is in C++ and is object oriented in nature. The entire VRTB is implemented as a set of classes in C++. This provides a generic way to implement many different kinds of players. Because an object oriented design was used, all of the attributes and functions which are common to Players and Ghosts are placed in the most abstract classes. Therefore, only the interface to these abstract classes will be described.

There are 4 major components to any one particular player. Each of these components will be implemented as a subclass to one the base (abstract) VRTB classes. The abstract classes are :

AbstractPlayer

This class defines the basic components of the simulation player. Methods in this are used to support such activities as processing incoming messages, internal state configuration and message creation.

AbstractLocalGhost

This class defines the components of the ghost object. An instance of this class is located in the AbstractPlayer, there are imbeded methods which use the ghost to check the state of remote ghosts. The function of the AbstractLocalGhost and the AbostractRemoteGhost (below) are the same, execpt that AbstractLocalGhost is the ghost which is located on the same machine as the player.

AbstractRemoteGhost

This class defines the view of a player by other remote players. When a simulation player is located on a remote workstation, its only line of communication is through remote ghosts. These objects contain limited state information which is useful to everyone. When the state of the player changes signifigantly from the dead reckoned state, a message is sent to all remote ghosts to reflect the new value of the state.

AbstractState

This class defines the state variable used in the AbstractPlayer. Each implementation will inherit from this class and use it only as a guide.

The main idea behind implementing each specific player as a separate subclass is to promote reuse. For example, there may be several different specific implementations of a simulation player. If there are several common attributes in these players, then it is useful to place then in a higher abstract class.

Another aspect of specific player implementation is to remember to conform to the protocol that has been established. In the following section, there will be methods that each player must implement. The behavior of each method will be described. It is important that these instructions be followed. The protocol will not work correctly if the state information is modified in the wrong places.

Implementation Details

This section gives details on how each class must be implemented. Along with the descriptions of the new classes, an example will be developed. This example is called MovingObj. It is not intended to have functional use, but is given to provide some insight to the coding process.

AbstractPlayer

The first class that must be created is a subclass of AbstractPlayer. There are three methods that must be reimplemented in the new player. They are **processMsg**, **internalProcess**, **createGhost** The player must also have a constructor method. The following is a formal template for the constructor method which is followed by the example method for the class MovingObj. The comments in the *(parenthesis)* are to be replaced by the implementer. Since this is a class definition, usually only method prototypes are given, so it will be useful to compare the template with the example. All methods defined in this document will be presented in this same format, with the template first and the specific example second.

```
class (your class name) : public (AbstractPlayer or any subclass of) {
        /* As with any C++ class, private methods and instance
        variables are allowed.  They should be placed in this section
        and unless otherwise stated, are allowed in all VRTB subcalsses */
public :
        /* Constructor method */
        (your class name) (char *);

        /* Reimplemented methods */
        void                    processMessage(Message);
        void                    internalProcess(void);
        AbstractGhost   *createGhost(void);

        /* Your methods go here */
};

class MovingObj : public AbstractPlayer {
public :
        /* Constructor method */
        MovingObj(char  *);

        /* Reimplemented methods */
        void                    processMessage(Message);
        void                    internalProcess(void);
        AbstractGhost   *createGhost(void);
};
```

Method : MovingObj(char*)

The purpose of constructor methods in C++ is to provide a default way to instantiate new instances of a class. In our case, a string is the required parameter, it should contain the name of the player. The main function of the constructor is to initialize the **state** instance variable. The following is the constructor template and the actual constructor for class MovingObj :

```
(your class name) :: (your class name)(char *name) : (name)
{
        (your state class name)          *s;

        /* Install an instance of your state */
        s = new (your state class name);
        state = s;

        /* Other initializations go here */
}

MovingObj :: MovingObj(char *name) : (name)
{
        MovingObjState          *s;

        s = new MovingObjState;
        s->acceleration.value(1.0, 2.0, -3.0);
        state = s;
}
```

The important things to note here are that the input parameter **name** is passed to the superclass, this is required. Also, the variable **state** is assigned as an instance of the class MovingObjState. The instance variable state is inherited from an abstract superclass. It must be initialized in this method. The class MovingObjState will be described later.

Method : void internalProcess(void)

This method serves two purposes. The first is to perform any internal processing which might be required by the player. The internal processing of the player is totally removed from the communications aspect of the VRTB. It is the decision of the player to decide what to put here, but this method will most likely compute the next state of the player.

The other purpose of this method is to update the state information. This is the only place in the player that the state information is allowed to be modified. Specifically, the instance variable **state** is allowed to be modified in this method. It is best to have this assignment as the last step of the method. The following is the template for the method internalProcess and the example for class MovingObj :

```
void (your class name) :: internalProcess(void)
{
        (your state class name)        *s;

        s = (your state class name *) state;
        /* Calculate state information and general processing */
}

void MovingObj :: internalProcess(void)
{
        float                   dt = 0.2;
        MovingObjState          *s;

        s = (MovingObjState *)state;
        s->velocity += s->acceleration * dt;
        s->location += s->velocity * dt;
}
```

Notice that the local variable **s** is needed. This is because of the typing done by C++. The instance variable **state** is implemented as a pointer to the class AbstractState, so the cast of state to a local version is necessary. It is important to follow this model, otherwise any dereferencing of state will lead to problems.

Method : createGhost(void)

This method is the standard way to create internal ghosts. There are some very specific instructions which must be followed in this method. It is easiest to look at the template and the example and explain later.

```
AbstractGhost *(your class name) :: createGhost(void)
{
        (your local ghost class name) *myGhost;
        char                                   ghostName[32];

        sprintf(ghostName, "%s_G", name);
        myGhost = new (your local ghost class name) (ghostName);
        return(myGhost);
}

AbstractGhost *MovingObj :: createGhost(void)
{
        MovingObjLocalGhost          *myGhost;
        char                         ghostName[32];

        sprintf(ghostName, "%s_G", name);
        myGhost = new MovingObjLocalGhost(ghostName);
        return(myGhost);
}
```

This method must return a pointer to the class AbstractGhost, regardless of the specific type of the actual local ghost. The name of the ghost is to end with "_G" (see the section **AbstractGhost** for details about the specific type of the local ghost).

Method : processMsg(Message msg)

This is probably one of the most confusing parts of the VRTB. It is still unclear if internal processing of messages should be implemented by the player itself. One of the main arguments against the use of C++ as the base language for the VRTB is that is does not fully support *polymorphism*. Dynamic lookup of method calls is restricted in some cases because of the strict type checking. Because the design of this project incorporates abstract classes and even abstract hierarchies, a language with run-time support of dynamic binding would be more sutable. Because of the evolving nature of this project, this subject should remain as a topic of discussion in the future.

Because C++ does not have dynamic binding of method calls, and it does not internally support inter-machine communications, a message system is necessary. Support for sending messages between players has been supplied, but the responsibility of actually sending the message is put on the implementers of VRTB players. Because this message sending is not a straightforward idea, a review of the message sending is given below.

- Player1 located on Machine1 wishes to send a message to Player2, not knowing that it is located on Machine2.

- A message is constructed and sent.

- The local router receives this message and determines that the first receiver should be the RemoteGhost of Player2 which is located on Machine1.

- The RemoteGhost of Player2 determines if it can answer the message directly (a query message), or must pass this message along to the actual player (a command message).
    - In the case of the query message, the RemoteGhost computes the answer to the message and send a message back to the sender of the original message.
    - In the case of the command message, the RemoteGhost reconstructs the message with the appripriate source and destination and then sends this message off.
- The local router again receives this messages and determines if the message is to remain local, or to send it to the router which is located on Machine2.
- At this point, either Player1 has received an answer to the message from the remote ghost or the router on Machine2 has received the message. The router located on Machine2 will then send the message to Player2 and it will respond accordingly.

To be able to send and receive messages, each player and ghost must have some method to encode and decode message. The VRTB protocol provides a standard method in which all incomming messages may be deocded. Sending messages is, again, left up to the discretion of the player. Sending a message will be discussed in the section **Message**.

When a message is recived by a player, the method **processMsg** is called. The input parameter, **msg** is an instance of the class Message. It has attributes (see section **Message** for more details) which can be accessed and will be useful in this method.

It is important to note that messages can come from many different places and from many different players, therefore, the format of all messages is same. To distinguish between the different kinds of message, there is a type field. To get the contents of the message, there will be a method which decodes the raw data and converts it into something meaningful.

In addition to the implementing this method, each new type of message that is added to your system must be placed in a header file. The name of this file is "data.h" and the new message is placed in the data structure MsgType in the file "data.h". The following is the template for the processMsg method :

```
void (your class name) :: processMsg(Message msg)
{
        switch(msg.type) {
                case (some mesage type) :
                                /*
                                        1. Decode the message
                                        2. Action
                                */
                                break;
        }
}
```

In the following example, MovingObj class must respond to the incomming messages. The message CHANGESPEED is a *command* and only the real player is able to respond to this command. It does so by decoding the message and changing is velocity. The message **VELOCITY** is a *query* of the player. If the gost is properly designed, query messages will never be sent to the player, only its ghost. Accepting the message VELOCITY in the player class is permitted and may be useful in debugging situations.

```
void MovingObj :: processMsg(Message msg)
{
        char            buf[25];

        switch(msg.type) {
                case STATEQUERY :
                                /* This is a query which should not have
                                   made it this far, but respond and print error */
                                state->getByString(buf);
                                Message newMsg(this->name, msg.src, SQANSWER, buf);
                                sendMail(newMsg);
                                printf("Error: STATEQUERY should not get this far\n");
                                break;

                case CHANGESPEED :
                                /* This is a command, decode message and
                                   change the velocity accordingly. */
                                getNextComponent(msg.msgBody, buf);
                                velocity = atof(buf);
                                break;

                case VELOCITY :
                                /* This is a query which should not have
                                   made it this far, but respond and print error */
                                velocity->getByString(buf);
                                Message newMsg(this->name, msg.src, VELOCITY, buf);
                                sendMail(msg);
                                printf("Error: VELOCITY should not get this far\n");
                                break;

                default :       break;
        }
}
```

AbstractGhost

There are two kinds of abstract ghosts, local ghosts and remote ghosts. The major difference between these two is that a local ghost is instantiated inside of the player whereas a remote ghost will run as a process on every machine other than the player's home machine. Many of the methods between these two are the same, so it is convenient to describe them both here. As a builder of simulation players, you will simply have to create two classes with the same methods in them. The class definition template and examples are given below :

```
class (your local ghost class name) : public (AbstractLocalGhost or
                                                any subclass of) {
public :
        (your local ghost class name) (char *);
        void    deadReckoning(void);
        void    processMsg(Message);

        /* Your methods and instance variables go here */
};

class (your remote ghost class name) : public (AbstractRemoteGhost
                                          or any subclass of) {
public :
        (your remote ghost class name) (char *);
        void    deadReckoning(void);
        void    processMsg(Message);

        /* Your methods and instance variables go here */
};

class MovingObjLocalGhost : public AbstractLocalGhost {
public :
        MovingObjLocalGhost(char *);
        void    deadReckoning(void);
        void    processMsg(Message);
};

class MovingObjRemoteGhost : public AbstractRemoteGhost {
public :
        MovingObjRemoteGhost(char *);
        void    deadReckoning(void);
        void    processMsg(Message);
};
```

Method :

MovingObjLocalGhost(char*)
MovingObjRemoteGhost(char*)

This method is a constructor for the class MovingObjLocalGhost and MovingObjRemoteGhost. As stated before, there must be one method which can create a instance of a particular class. The main function of this constructor is to instantiate the ghosts. The following is the template for this method and an example from the MovingObj player :

```
(your local ghost class name) ::
                    (your local ghost class name) (char *name) : (name)
{
        state = new (your state class name);
        newState = new(your state class name);

        /* Other ghost initializations go here */
}

MovingObjLocalGhost :: MovingObjLocalGhost(char *name) : (name)
{
        state = new MovingObjState;
        newState = MovingObjState;
}
```

Any other initialization of specific local variables should also go in this method.

Method : deadReckoning(void)

The only objective of this method is to let the Ghost compute its limited state model. It is up to the ghost to determine the next state of the player, without any additional information coming from the player. The following shows the dead reckoning algorithm template method as well as the example for the MovingObj player :

```
void (your local ghost class name) :: deadReckoning(void)
{
        (your state class name)         *s1; *s2;

        s1 = (your state class name *) state;
        s2 = (your state class name *) nextState;

        /*
                State computations using local variables s1 and s2.
                Usually, s2 is computed using information located in s1.
        */
}

void MovingObjLocalGhost :: deadReckoning(void)
{
        float           dt = 0.2;
        MovingObjState          *s1; *s2;

        s1 = (MovingObjState *)state;
        s2 = (MovingObjState *)nextState;
        s2->orientation = s1->orientation;
        s2->acceleration = s1->acceleration;
        s2->velocity = s1->velocity + s1->acceleration * dt;
        s2->location = s1->location + s2->velocity * dt;
}
```

There are a few aspects of this method to notice. First, the local instance variables **s1** and **s2** are assigned the class instance variables state and **nextState**. When accessing these two class instance variables, this

Method : processMsg(Message msg)

Please refer to the AbstractPlayer, processMsg method for justification on why this is a good way to handle messages. Here you will find an example of the MovingObj method. The template can be found in Abstract Player.

This example is for the local and remote ghost. In this case of processMsg, the message **CHANGESPEED** is not to be handeled by the ghost, so it must always pass this message onto the player. The message **VELOCITY** is a query and can be answered by this ghost. This is the kind of message that should not be passed along to the player. The following is the example method processMsg from the ghost classes of MovingObj :

```
void MovingObj :: processMsg(Message msg)
{
        char            buf[25];

        switch(msg.type) {
                case STATEQUERY :
                                sendUpdateMsg();
                                break;

                case CHANGESPEED :
                                /* This is a command, so send it along */
                                msg.src(this->name);
                                sendMail(msg);
                                break;

                case VELOCITY :
                                /* This is a query, so answer it */
                                velocity->getByString(buf);
                                Message newMsg(this->name, msg.src, VELOCITY, buf);
                                sendMail(msg);
                                break;

                default :       break;
        }
}
```

AbstractState

To let the user of VRTB have the most accessibility to the state information, an abstract state was created, called AbstractState. This abstract class provides default definitions of methods that must be reimplemented. It defines no instance variables. This means that the user must define and maintain all of his own instance variables.

There is one aspect to take note of when implementing subclasses of AbstractState. This problem arises because C++ is not completely polymorphic. Essentially, while the message passing paradigm ensures that a choice of method is found based on the class of the receiver, the selection of a method to be used in an overridden operator is based on the declared (static) types of the arguments and not the current (dynamic) types, which the VRTB to work correctly. Simply place all normal initialization for the class in this method.

```
(your state class name) :: (your state class name)(void)
{
        /* Any state initialization goes here */
}

MovingObjState :: MovingObjState(void)
{
        location.value(0.0, 0.0, 0.0);
        velocity.value(0.0, 0.0, 0.0);
        acceleration.value(0.0, 0.0, 0.0);
        orientation.value(0.0, 0.0, 0.0);
}
```

It is important to remember that all of the classes described here are part of a class heirerchy. When designing interesting player behavior, it will be useful to design levels of abstract and concrete classes. The following is an example of a class definition and constructor subclass of MovingObj called Vehicle. The class definition is similar to the ones seen before, but the constructor has been modified to show inherited initialization.

```
class VehicleState : public MovingObjState {
public :
        /* A vehicle has numberOfDoors, and engineType */
        int     numberOfDoors, engineType;

        /* These methods are reimplementations of higher level ones */
        VehicleState(void);
        void    operator=(AbstractState *);
        int     operator==(AbstractState *);
        int     operator!=(AbstractState *);
        int     setByString(char *);
        int     getByString(char *);
        void    print(void);
}

VehicleState :: VehicleState(int nod, int et, Vector l, v, a, o) : (l, v, a, o)
{
        numberOfDoors = nod;
        engineType = et;
}
```

The technique of passing parameters to the parent class in initialization can be done in any class/subclass pair. The MovingObj class must have a constructor which will acccept these parameters as well. This code is given below.

```
MovingObjState :: MovingObjState(Vector l, v, a, o)
{
        location.value(l.x, l.y, l.z);
        velocity.value(v.x, v.y, v.z);
        acceleration.value(a.x, a.y, a.z);
        orientation.value(o.x, o.y, o.z);
}
```

Method : operator-(AbstractState *)

The input parameter is received as a pointer to AbstractState. Before any use of this input, it must be converted to a pointer which conforms to the current state. In our example, this parameter will be converted to a pointer to MovingObjState. Once this conversion is done, then it can be used as a normal state variable. The code below shows the template for this method as well as the implementation of the assignment statement for the class MovingObjState :

```
void (your state class name) :: operator-(AbstractState *state)
{
        (your state class name)         *tmp;

        tmp = (your state class name * ) state;
        /* Assignment of tmp to local instance variable */
}

void MovingObjState :: operator=(AbstractState *state)
{
        MovingObjState          *tmp;

        tmp = (MovingObjState *)state;
        location = tmp->location;
        velocity = tmp->velocity;
        acceleration = tmp->acceleration;
        orientation = tmp->orientation;
}
```

Notice the two requirements and how they are used in this method. First, the method argument is defined to be a pointer to the class AbstractState, second, the first action of the method is to convert the argument to the type of the class. These two operations must be done for each inhered function in the concrete state classes.

Method :

operator--(AbstractState *)
operator!-(AbstractState *)

This is two methods are so similar in functionality that is worth while to describe them both here. Again, the rules from above still apply. Their implemetations are straight forward and are given below as well as a template for the boolean operators.

```
int (your state class name):: operator (boolean operator)
                                                (AbstractState *state)
{
        (your state class name)         *state;

        tmp = (your state class name*) state.
        /*
                Compute the value of the boolean operation
                and return the result
        */
}

int MovingObjState :: operator==(AbstractState *state)
{
        MovingObjState          *state;

        tmp = (MovingObjState *)state;
        if(location==tmp->location &&
                velocity==tmp->velocity &&
                acceleration==tmp->acceleration &&
                orientation==tmp->orientation)
                return(1);
        else
                return(0);
}

int MovingObjState :: operator!=(AbstractState *state)
{
        MovingObjState          *state;

        tmp = (MovingObjState *)state;
        if(location!=tmp->location &&
                velocity!=tmp->velocity &&
                acceleration!=tmp->acceleration &&
                orientation!=tmp->orientation)
                return(1);
        else
                return(0);
}
```

Method : setByState(char *)

This method is used by the semi-automatic messaging system. It provides a way for the user to decode state information which has been sent via VRTB message. In the case of the MovingObjState class, we have provided default setbyString messages which can be sent to instance of the class Vector. In general, users will provide methods which will encode types into strings and decode strings into types. The only requirement is that each message follow the standard format (see section **Message** for more information). The following is the **setbyString** and **getByString** methods which have been implemented for the class MovingObjState :

```
int (your state class name) :: setByString(char *msgBody)
{
        char   *p;

        p = msgBody;
        /*
                Decode the string msgBody in the
                same order it was encoded
        */
}

int (your state class name) :: getByString(char *msgBody)
{
        char   w[64];
        int     len=0;

        /*
                Encode each state variables in some specific order
                into the string msgBody, return 0 if any problems occur
        */
}

int MovingObjState :: setByString(char *msgBody)
{
        char    *p;

        p = msgBody;
        if(location.setByString(&p)==0) return(0);
        if(velocity.setByString(&p)==0) return(0);
        if(acceleration.setByString(&p)==0) return(0);
        if(orientation.setByString(&p)==0) return(0);
        return(1);
}

int MovingObjState :: getByString(char *msgBody)
{
        char    w[64];
        int     len=0;

        location.getByString(w);
        sprintf(msgBody, "%s|", w);
        velocity.getByString(w);
        sprintf(msgBody, "%s|", w);
        acceleration.getByString(w);
        sprintf(msgBody, "%s|", w);
        orientation.getByString(w);
        sprintf(msgBody, "%s|", w);
        return(strlen(msgBody));
}
```

As you can see, it is up to the user to define how the strings are constructed, including the order in which the variables are placed in the string. Because state information is needed at the abstract levels, the reimplementation of these two methods are required.

Message

The class Message provides a way for the user to send C++ messages from one object to another. One of the parts of this project that has not yet been clearly defined is how to properly perform the message sending. It is possible that G++ will take care of this task for us?

We have provided this class for the implementers of players who are using the VRTB protocol. It will probably not be permanent, but for now it will have to work. There are some uses of messages in the examples already given, please refer to these examples for help. Additionally, all of the source code for the Message is provided.

APPENDIX C

Published Paper : 1992 Symposium on Interactive 3D Graphics

Computer Graphics 26:2, March [illegible]
Special Issue on 1992 Symposium on
Interactive Cptr Graphics

Networked Virtual Environments

Brian Blau, Charles E. Hughes,
J. Michael Moshell and Curtis Lisle

Institute for Simulation and Training
and
Department of Computer Science
University of Central Florida
Orlando, Florida 32816

ABSTRACT

The Virtual Environment Realtime Network (VERN) is an object oriented testbed for the interconnection of environments over a network of graphical workstations. VERN is based on extensions to the networking technology of the DARPA sponsored SIMNET combined combat training system and the Distributed Interactive Simulation protocol being developed as a DOD standard. It allows for multiple participants to interact in an environment, sharing ideas and solving problems, regardless of their physical locations. Furthermore, dramatic reconstructions of historical events for education or entertainment will be possible. Indeed, much of the impact of VERN is likely to result from the ability of participants to learn from each other even if they and their machines are separated by long distances.

INTRODUCTION

Virtual Reality/Virtual Environments (VE) describes a multi-sensory real-time simulation that immerses the participant in a multi-dimensional (usually 3D) graphical space, allows freedom of movement within the space, and supports interactions including the modification of most features of the space itself [10,13]. Additionally, a VE system may include modeling tools for world construction, rendering tools for viewing, storage mechanisms for saving memorable experiences, I/O devices for controlling aspects of the space and communication ports for shared environments.

Recently, research in the VE field has now turned its attention to networking issues for shared experiences. Two phases must be considered : rendering (distribution of graphical data) and computation (distribution of the physical model). The Visual Systems Laboratory (VSL) at IST is currently working on both of these problems.

Our efforts have produced two software systems : ANIM and VERN. ANIM is an interactive graphical simulation system with support for devices like SpaceBalls and gloves (VSL Input Paw). Modeling tools, such as Alias (high end rendering tool, Alias Research), MultiGen (tool for CIG databases, Software Systems) and S1000 (SIMNET's CAD system, BBN) are used to build environments which are inputs for the system. ANIM has been extended using VERN protocols and can now operate on several computers, distributing the computations of the objects as well as distributing the space itself. This paper will focus on VERN and how systems like ANIM can use VERN to distribute virtual objects, computational load and user interactions across multiple simulation platforms.

Simulation Network (SIMNET) [7,12] is a project sponsored by the Defense Advanced Research Projects Agency (DARPA) and was designed and built by BBN Laboratories Inc. and Perceptronics Inc. It allows for collective team training in combined arms scenarios. All of the simulators are networked via EtherNet and the communication model is based on the "dead reckoning" paradigm [8]. VE applications are a far more demanding simulation than SIMNET, because in a truly useful virtual world, every object is dynamic. In traditional simulators, only a small collection of moving objects can be maintained.

As a follow-on to the homogeneous SIMNET system, the US Army has explored the possibility of expanding these concepts to address the networking of large numbers of dissimilar training devices. The next important step in this research is the development of a standard communications protocol for Distributed Interactive Simulations (DIS) [8].

Interactive simulations in the SIMNET and DIS worlds perform computations and communicate by a dead reckoning model. Each object in the simulation has a host machine which will process its dynamics. All other machines have representations of the object which maintain an approximation to the current state of the object. The approximation of a simulation object's state is computed by a dead reckoning algorithm. This computation is usually an extrapolation of the object's position based on velocity. When the host object realizes that the dead reckoning model has deviated significantly from the dynamic model (probably because of user input), an update message is sent to all other representations of the object on every other machine.

DESCRIPTION OF VERN v1.2

VERN v1.2 was developed to meet the needs of the simulation community as a vehicle for development of networked environments as well as to break new ground in the development of interactive VE systems. This implementation is an extensible object oriented class hierarchy where the communications, dead reckoning and process control are abstracted to the highest levels. Most importantly, VERN extends the notion of dead reckoning into a distributed physical model.

VERN evolved from a non-realtime Smalltalk-80 prototype [2,3,4]. Version 1.2 is implemented in C++ and currently runs on Silicon Graphics and Sun Sparc UNIX systems.

The communications protocol forms the software basis for an environment that will support experiments with a network of visual simulators operating in a single simulation. This environment will contain dynamic and static objects. For example, terrain over which objects move may be dynamic while buildings in a city may be static. Objects in the simulation communicate with each other without having to know the host machine on which the receiving object resides. Each object assumes that all objects are in its own local memory. Under the VERN protocol, messages bound for remote objects are intercepted and routed accordingly.

Players and Ghosts

Each real world object participating in the simulation is represented by a software object called a Player. The Player resides on the object's home machine. If human or external input is required by the Player, the data is read and processed on the Player's home machine. The main responsibility of the Player is to accurately maintain state information, read and process inputs, provide feedback usually in the form of real-time graphics, and inform the network of any significant state changes that deviate from the dead reckoning model.

In order to facilitate communication between Players residing on separate machines, each Player has an associated Ghost located on every machine involved in the simulation. Thus in an N Player simulation on M networked machines, each machine is guaranteed to have exactly N objects representing all players. Such a configuration allows Players to communicate locally with any other Player (represented by its Ghost). It is the responsibility of the Ghost either to respond directly to the message, or to forward it to the actual Player.

Ghosts are approximations of their associated Players. That is, the state of a Ghost is not always as precise (algorithmically) as the Players, but this approximation is adequate for visualization and dynamics. All Ghosts that are associated with a single Player are synchronized at any given instant in simulation time through the use of the system clock, message passing and dead reckoning. When the Player realizes that its Ghosts are going to be inaccurate, the Player then communicates the correct state information to all Ghosts.

Message Types

There are two types of messages to which Players and Ghosts respond: queries and commands. Queries are messages which can be processed entirely by the Ghost. Commands are messages that must be passed on to the Player. Thus, a message that requests state information would be considered a query while a change of behavior message would be a command.

Class Hierarchy

VERN v1.2 was designed using the object oriented paradigm. The classes that comprise the highest levels of the hierarchy contain the code for handling all of the communications and process control protocols. This hierarchy is considered a white box framework [6] because the user (programmer) of the system must follow the structures that the abstract classes establish. Figure 1 shows the abstract class hierarchy of VERN v1.2.

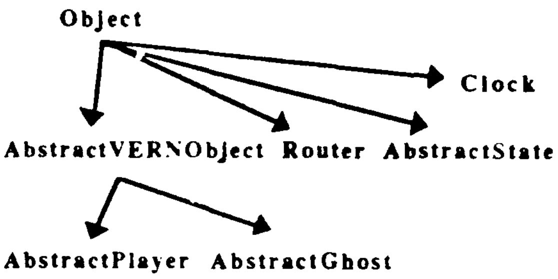


Figure 1. Class Hierarchy for VERN v1.2

There are additional classes not shown here which represent communication support structures such as mailboxes, addresses, and sockets. The following describes each of the abstract classes.

class AbstractVERNObject :

This class contains the virtual methods which handle actions to be performed in each simulation loop. For example, initializations, maintenance of the local mailbox (repository for messages), and access to the state information.

class AbstractPlayer :

This class defines the basic components of the simulation Player. Virtual methods in the class are used to support such activities as processing of incoming messages, internal state configuration and message creation.

class AbstractGhost :

This class defines the "view" of a Player as seen by other local and remote Players. A simulation Player located on a workstation can communicate with another Player only through its AbstractGhost. An instance of this class contains limited state information which is useful to other Players. When the state of the Player changes significantly from the dead reckoned state, a message is sent to all AbstractGhosts to reflect the new value.

class AbstractState:

This class defines the state variables used in the AbstractPlayer. Each implementation will inherit from this class and use it as a guide. The class AbstractVERNObject has an instance of AbstractState as one of its instance variables.

IMPLEMENTATION DETAILS

To write a Player/Ghost program, the programmer must create concrete subclasses of the abstract classes listed above. For example, consider the definition of a moving ball. The classes that must be created are MovingBallPlayer (subclass of AbstractPlayer), MovingBallGhost (subclass of AbstractGhost), and MovingBallState (subclass of AbstractState). These new classes must then be compiled, linked and executed. Further examples of Players may be found in [2].

The first classes that must be created is a subclass of Abstract Player and AbstractGhost. There are two methods that must be reimplemented in the new Player. These are processMsg and computeNextState. The Player must also have a constructor method to create instances.

Method : constructor

The purpose of constructor methods in C++ is to provide a default way to instantiate new instances of a class. In our case, a string containing the name of the Player is the required parameter. The main function of the constructor is to initialize the state instance variable.

Method : processMsg

Since C++ does not internally support machine to machine communications, a low level messaging system is necessary. Support for sending raw packets of data between UNIX processes has been supplied. The responsibility of creating and interpreting the raw data is left to the Player.

The purpose of processMsg is to interpret and respond to incoming messages. It is important to note that messages may arrive from many different Players. Each raw message contains the source, destination, data and type.

Method : computeNextState (for Player)

This method serves two purposes. The first is to perform any internal processing which might be required by the Player. For example, calculate new position and velocity based on current simulation time. The second purpose of this method is to update the state information of the Player.

Method : computeNextState (for Ghost)

The objective of this method is to compute the Ghost's approximate state model. The Ghost determines the next state of the player, without any additional information coming from the player. This is how dead reckoning is implemented within VERN. Each Ghost performs this message once each simulation loop.

In order to facilitate complete freedom in defining state information of a Player's object, an AbstractState was created. This abstract class provides default definitions of methods that must be reimplemented. It defines no instance variables. This means that the concrete Player class must define and maintain all of its own instance variables. The main methods in this class are comparison operators such as == and !=, mathematical operators such as + and -, and the assignment operator =. There are no other restrictions placed on the addition of subclassses.

EXECUTING THE SIMULATION

Previous versions of the VERN used a synchronized clock as the simulation coordinator. Using this synchronization system enabled the state of the Player to know (via a local dead reckoning) the Ghost's exact state at every tick of the clock. Although this is important, it can be accomplished using the computers' real-time clocks. This allows each computer to execute as fast as possible and it also reduces the communications overhead of clock maintenance.

The function of the Router is to maintain the connections to the outside world, maintain a list of active local and global Players, and route messages according to their source and destination. All of the routers know the locations of the other routers and the addresses of all objects. This global information allows the router to make decisions about the direction of the message. The Router's main loop asks each of the local objects to run one simulation cycle. During this cycle, objects execute the inherited methods above.

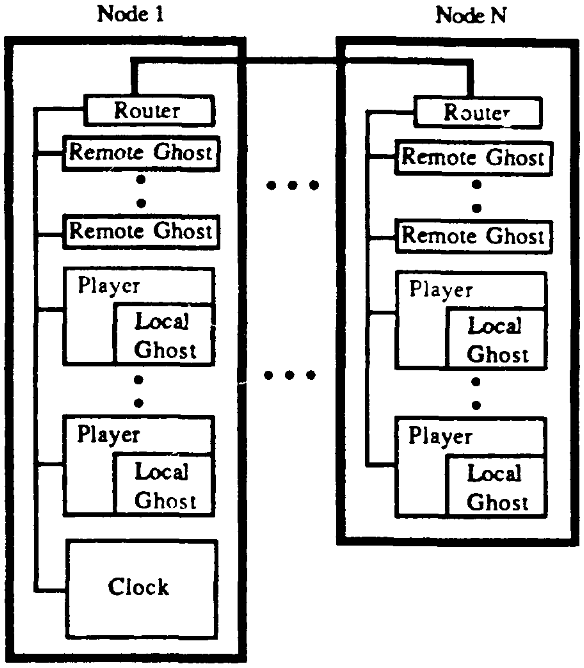


Figure 2. Process Architecture of VERN v1.2

Additionally, the Router can report the current simulation configuration and detect simulation errors. When a Player leaves the simulation, the Router immediately realizes which Player is missing and then reports this to all Routers in the system. Figure 2 shows the overall system architecture of VERN v1.2.

There are additional system functions worth mentioning. An automatic update is has been added. This forces the Player to update its Ghost at a specified interval (usually 3-5 seconds) even if no update is needed. This function is useful when the communication system drops packets. Using this function provides for reliable Player/Ghost synchronization.

An additional system parameter is called "dynamic update." Dead reckoning algorithms have a base threshold on which an update is based. The dynamic update is another threshold which provides the user with some control. The dynamic update threshold specifies the amount of error in the dead reckoning algorithm. For example, if the user is interacting with the environment at a detailed level, then the dynamic update will be set to a small value, resulting in accurate synchronization between Player and Ghost.

One last feature is called "update tracking." When a Ghost receives an update message from the Player, usually the position has changed significantly. If the update tracking is set to "jump", then the object will disappear from its current location and reappear at its updated location, causing a visual disturbance. If the update tracking is set to smooth, then the object will track evenly to its new position. This tracking will occur over a number of frames and the amount of smoothing can be set as a system parameter.

ISSUES FOR DISCUSSION

There are many issues that arise in research projects of this nature. It is useful to note that VERN v1.2 is only one part of a larger project to develop VEs, and its main purpose is to show proof of concept. Below are a few interesting topics that emerged from this implementation.

Communications

The base communications between different machines is accomplished with a "broadcast" UNIX socket. Broadcast sockets distribute their packets to anyone listening. Socket communications using the broadcast mechanism are not guaranteed: messages sent may not reach their final destination. Additionally, broadcast is convenient for local communications but may not be useful in long haul systems. By experiment, the performance advantages of broadcast messages outweigh the risk of occasional lost messages. Point-to-Point sockets are the main means for long haul communication. VERN has been tested over private communication lines as well as on the nation-wide Internet.

It should be noted that the lowest level communications were written in-house. There was a study of language based communications systems, such as those that support TimeWarp and Actor [5,1]. It was determined that these systems are useful, but our need to learn and experience workstation based communications outweighed their use. Implementing VERN using an Actor or TimeWarp paradigm is possible and is part of our future research.

Object Oriented Design Using C++

This is probably one of the most interesting parts of the VERN. One of the main arguments against the use of C++ as the base language for the VERN is that is does not fully support polymorphism. Dynamic binding of method calls is restricted in some cases because of the strict type checking. Since the design of this project incorporates abstract classes, a language with flexible support of dynamic binding and type checking would be more suitable. Smalltalk would be a suitable alternative and may solve some of these problems, but it is not yet available on a wide variety of workstations.

Performance

The performance of VERN has been measuerd and a detailed description of experiments can be found in [3]. Currently, running on a network of 2 Sun Sparcs and 4 Silicon Graphics workstations, VERN v1.2 can achieve 300-350 frames/second. The test environment consisted of 5 balls bouncing in a closed box. We have conducted extensive experiments on non-trivial environments and the results are encouraging. We expect frame rates of 5-10 per second with an environment consisting of 1000 objects (~10k polygons).

Future Directions of VERN

The major goals of the next version are to improve efficiency, investigate other distributed simulation systems, experiment with extended environments and continue work on long haul communications.

The design of this project represents only one limited view of VE system development. Frameworks, like VERN, need to be combined with other object oriented systems to form complete VE systems. This project and others, like ANIM, are likely to pave the way to robust systems. These new VE's will contain physical modeling, real-time control of objects, decentralized clocks and spatial division of computations in an object oriented framework. The next level of research for this project will look at these issues to determine commonality and reusability which will extend the functionality of the entire system.

ACKNOWLEDGEMENTS

Many thanks are due to Jinxiong Chen and Xin Li for their excellent ideas and late nights. Also, thanks goes to Ernie Smart for his continuing support. This research project is supported by the Army's Project Manager for Training Devices, Department of Defense, contract no. N61339-90-C-0041.

REFERENCES

[1] Agha, G., *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press 1986.

[2] Blau, B. "How To Implement Networked Simulation Players Using VERN v1.05." VSL Document 91.4. Institute for Simulation and Training, University of Central Florida, Orlando, FL. April, 1991.

[3] Blau, B., "Performance Measurements of VERN v1.2," VSL Document 91.37, Institute for Simulation and Training, University of Central Florida, Orlando, FL. Jan, 1992.

[4] Hughes, C. E., Blau, B., et. al, "Dynamic Terrain Project: The Virtual Reality Testbed Smalltalk Prototype." VSL Document 90.14, Institute for Simulation and Training, University of Central Florida, Orlando, FL, Nov, 1990.

[5] Jeffferson, D., et. al., "Distrubuted Simulation and the Time Warp Operating System." 11th ACM Symposium on Operating Systems Principles, Austin, TX, In, Operating Systems Review, vol. 25, no. 5, p77-93, 1987.

[6] Johnson, R. E and Foote, B., "Designing Reusable Classes." *Journal of Object Oriented Programming*, 22-35, June/July, 1988

[7] Johnston, R. S., "The SIMNET Visual System." *Proceedings of the 9th ITEC Conference*, Washington D.C. Nov, 1987.

[8] McDonald, L. B. and Bouwens, C. 1991. "Rationale Document for Protocol Data Units for Entity Information and Entity Interaction in a Distributed Interactive Simulation." Institute for Simulation and Training Publication IST-PD-90-1-Revised, University of Central Florida, Orlando, FL. (Jan)

[9] Moshell, J. M., et. al, "Networked Virtual Environments for Simulation and Training," *1991 International Simulation Technology Conference*, Orlando, FL, Oct, 1991.

[10] Pentland, A. P., "Computational Complexity Versus Simulated Environments," Special Issue on 1990 Symposium on Interactive 3D Graphics, In *Computer Graphics*, vol. 24, no. 2, March 1990.

[11] Pope, A., "The SIMNET Network and Protocols." BBN Report No. 7102. BBN Systems and Technologies Advanced Simulation Division, Cambridge, Mass. July, 1989.

[12] Pope, A., "The SIMNET Network and Protocols." BBN Report No. 7102. BBN Systems and Technologies Advanced Simulation Division, Cambridge, Mass. July, 1989.

[13] Rheingold, H., *Virtual Reality*, SummitBooks, 1991, ISBN 0-671-69363-8.